# Geospatial Phrase Grounding and Disambiguation

by

Amy Michelle Slagle

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2008

© 2008 Amy M. Slagle. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Author.....

Department of Electrical Engineering and Computer Science

May 27, 2008

Certified by.....

Michael Cleary

Charles Stark Draper Laboratory

Technical Supervisor

Certified by.....

Boris Katz

Principle Research Scientist

Thesis Advisor

Accepted by.....

Arthur C. Smith

Professor of Electrical Engineering

Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

NOV 13 2008

LIBRARIES

ARCHIVES

# Geospatial Phrase Grounding and Disambiguation

by

Amy Michelle Slagle

Submitted to the Department of Electrical Engineering and Computer Science

May 27, 2008

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

## Abstract

GeoCoder is a spatial reasoning system that converts natural language inputs into a set of precise spatial coordinates to display on a map. GeoCoder's spatial knowledge is represented in a set of ontologies. GeoCoder parses input phrases and adds location reference individuals to its ontology model. Relationships between location references are recognized based on mid-level structural patterns in the parsed phrase. GeoCoder grounds (or finds possible geometries for) location references in an iterative process, in which locations are grounded based on their relationships to previously grounded locations. GeoCoder improves upon previous systems by grounding and disambiguating at the phrase level, interpreting parses with rules that match mid-level structure patterns, expressing disambiguation heuristics in ontologies, and improving scalability by separating grounding from reasoning about relationships.

Technical Supervisor: Michael Cleary

Title: Decision Systems Group Leader, Charles Stark Draper Laboratory

Thesis Advisor: Boris Katz

Title: Principal Research Scientist

## Acknowledgements

I would like to thank my supervisor, Michael Cleary (Draper), and my advisor, Boris Katz (MIT), for their guidance and support in the research and writing of this thesis. Michael Cleary and Erik Antelman (Draper) conceived the initial vision for GeoCoder and developed ideas in frequent project meetings. I would also like to thank Erik Antelman for compiling data from the US Census Bureau's TIGER database into a PostgreSQL database for GeoCoder, and for developing a GeoCoder client that is integrated with Google Earth. Finally, I would like to thank Gary Borchardt (MIT) for his comments on the proposal and thesis draft, and Yechezkal Gutfreund (Draper) for developing a basic GeoCoder client in VB and contributing to the design of the client-server interface.

This thesis was prepared at the Charles Stark Draper Laboratory, Inc., under the GeoCoder Internal Research and Development Project 21799-001.

Publication of this thesis does not constitute approval by Draper of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

.....

Amy M. Slagle

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Spatial references in natural language rarely include detailed addresses or coordinates. Instead, they include potentially ambiguous place names and spatial prepositions that locate one place relative to another. People rely heavily on both implicit and explicit contextual information and knowledge of the most common senses of location names when interpreting these references. Automated spatial language processing systems face the difficult task of translating a vague spatial expression into a grounded location in space. This thesis focuses on the problem of converting a spatial expression like "the south of Boston" into a set of latitude and longitude coordinates that can be displayed on a map.

Existing geographic information systems (GIS) and gazetteers have compiled large amounts of data about places, including alternative names, latitudes and longitudes, and populations. For example, the GeoNames database [1] has over 8 million geographical names, including 2.2 million populated places with latitudes, longitudes, elevations, populations, and names in multiple languages. The US Census Bureau's TIGER database [38] includes shape files that represent the cartographic boundaries of places, such as cities, counties, and school districts. MetaCarta's gazetteer database [4], the Alexandria

Digital Library's gazetteer [17], and the Getty Thesaurus of Geographic Names [18] contain other useful geographic data. Gazetteers also include additional information such as population and elevation, and taxonomies of thematic classifications such as "populated place" and "country". Given a place name, gazetteers can be used to look up additional information that makes it easier to interpret the surrounding text.

Geographic information systems are necessary for grounding a spatial reference, but they are not sufficient. The first problem is disambiguation. Searching for "Boston" in the GeoNames database returns 676 raw results, spread across six continents. Many algorithms choose the best result heuristically, based on proximity to other locations mentioned in the discourse or the sizes and populations of the results [20]. A spatial reasoner could provide a disambiguation algorithm that considers specific relationships described in the text. The second problem is that places are often described through relative references rather than names. To interpret a phrase like "the south of Boston," a system needs to look up the location of "Boston" and figure out the meaning of "the south of." This leads to additional ambiguity problems. Even if GeoCoder can choose a single meaning of "Boston," spatial prepositions and direction words can take on different meanings in different contexts. Based solely on the individual words "south" and "of," we can't differentiate a region south of Boston from a region in the south of Boston. Sometimes it helps to disambiguate the spatial preposition and the location name together. For example, in the phrase "along Boston," "Boston" is unlikely to refer to a city, but might refer to a road. An additional complication is that a spatial reference can include arbitrarily long chains of prepositional phrases, with the number of possible parse tree nestings increasing exponentially.

GeoCoder restricts queries to a fixed set of spatial prepositions and direction words, though future work could incorporate a larger set of prepositions or spatial verbs. Some simple queries that the

system can resolve are "in X," "near X," and "in the south of X," where "X" is a location name that can be grounded with a GIS. These queries involve determining the location of one region based on the location of a single reference region. The system also handles queries with nested prepositions, such as "the park near the lake in the city south of Boston." GeoCoder also allows queries with two location names combined with "and", such as "the city between Boston and New York." More complicated queries with an arbitrary combination of location names and additional prepositions will be allowed in future versions. For example, these queries include "from X to Y," "between X, Y, and Z," and "in X or Y." Finally, some difficult queries contain anaphora instead of location names. An example is "near the city," where "the city" is an anaphor that refers to a city mentioned earlier in the discourse.

This thesis describes the development of a spatial reasoning system that is capable of interpreting spatial references involving combinations of location names and relative references. The GeoCoder system has a client-server architecture. The client sends a natural language query to the server, and the server sends a reply in GML format [26], which can be displayed by software like Google Earth [15]. The server uses a set of spatial ontologies to translate a natural language parse tree into a combination of queries that perform geometric functions or database searches, to determine a set of precise spatial coordinates. The server draws on a number of existing technologies, including the Stanford Parser [37] and Named Entity Recognizer [36], the Pellet reasoner [35], the Jena reasoner interface [8], and Java Topology Suite [40], and the PostGIS database extension [2] to the PostgreSQL relational database system [29]. The GeoCoder system described in this thesis is the first implementation of an idea developed under internal research and development funding at the Charles Stark Draper Laboratory, with the goal of intelligently extracting precise geospatial information from natural language. The general approach in designing the system has been to use existing tools whenever possible, and focus efforts on open research questions. This thesis describes the server component of

the GeoCoder system, which converts natural language into precise coordinates. The user interface and display engine are important aspects of the client component, but will not be addressed here.

Chapter 2 discusses background information and work related to the core components of the server. Chapter 3 describes the spatial ontologies and the issues and trade-offs involved in their development. Chapter 4 describes the design of the GeoCoder server, including the existing technologies used. Chapter 5 gives a set of representative test cases and the details of the server's processing. Chapter 6 describes open research questions and future extensions that could improve the system. Finally, Chapter 7 summarizes the GeoCoder system and the major contributions in this thesis. Figure 1.1 shows the steps in GeoCoder's processing flow, and the tools and data used at each step.

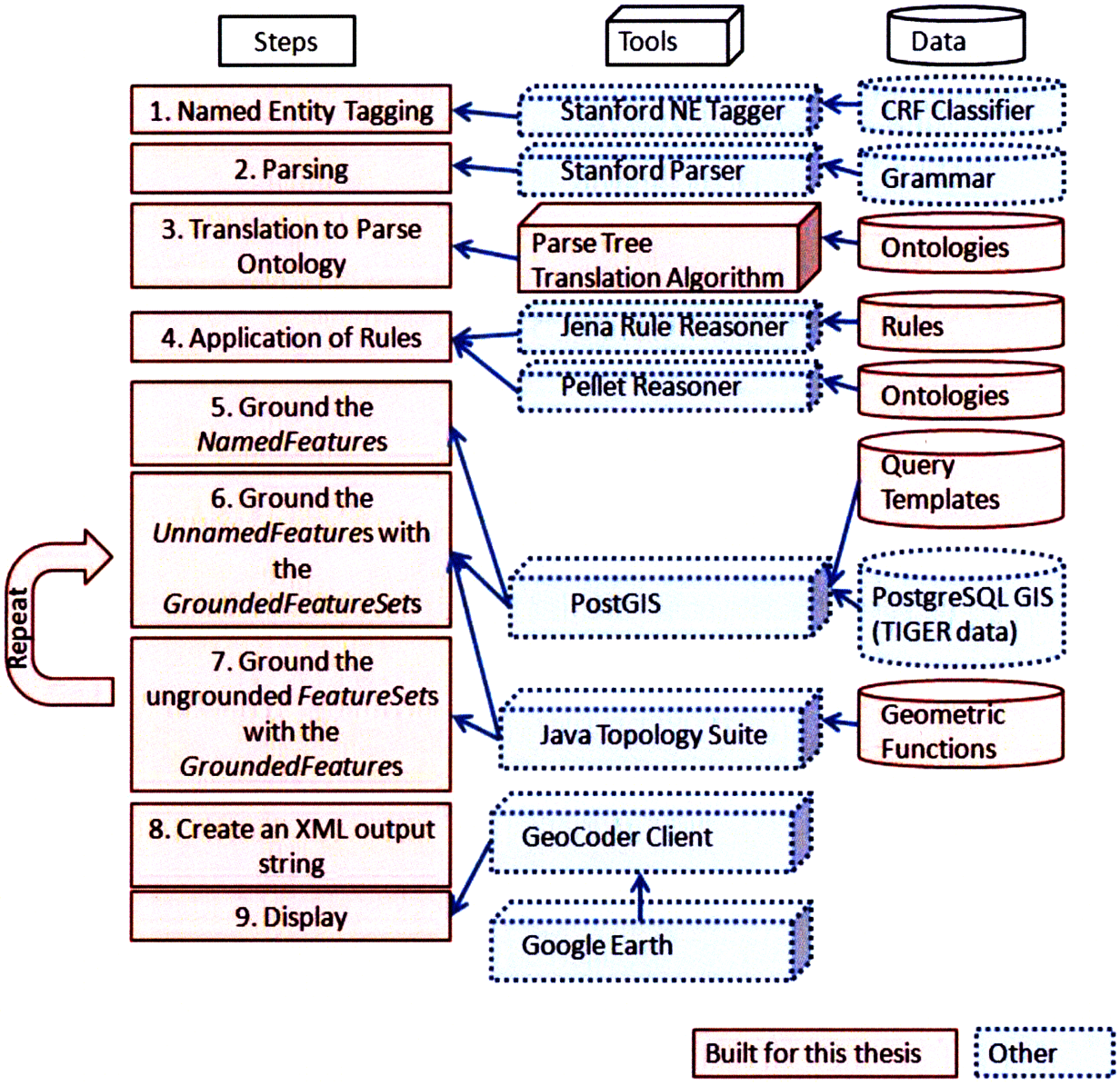


**Figure 1.1 GeoCoder Processing Steps**

# Chapter 2

# Background and Related Work

This chapter describes other research relevant to GeoCoder, and is divided into Background and Related Work.

## 2.1. Background

The Background section covers common approaches to basic problems faced by components of the GeoCoder system. Some of these systems are used by GeoCoder.

### 2.1.1 Recognizing Location Names

Given an input text, a spatial language processor's first task is to determine which of the words could refer to locations. Location references can be proper names (e.g. "Boston"), common nouns with relationships to other locations ("e.g. "the city near Boston"), anaphora that refer to previously

mentioned locations (e.g. "Boston ... the city"), or common nouns that refer to specific locations based on context (e.g. "my house"). GeoCoder only handles proper names and nouns with relationships to other locations. To determine which common nouns refer to locations, GeoCoder uses domain knowledge encoded by humans. Determining which proper names refer to locations is a sub-problem of the named entity recognition (NER) task, in which a system tags words that refer to particular types of named entities, such as location, person, or organization. Given the sentence "MIT is in Cambridge, Massachusetts," an NER system should tag "MIT" as an organization and "Cambridge" and "Massachusetts" as locations. One simple approach is to look up every word in a geographical dictionary (called a gazetteer) that lists place names and other geographic information, such as latitude/longitude coordinates and population. This is useful for determining candidate place names, but when used alone it can lead to many errors [41]. If the gazetteer is incomplete, it may fail to match some place names, or it may fail to match alternate spellings. It might also result in too many candidate place names, since some common English words also appear as foreign place names (e.g. the city named "Of" in Turkey or the city named "To" in Myanmar [3]). Other methods can reduce the number of candidate names to be checked against the gazetteer.

Some systems have achieved high performance on the NER task by creating hidden Markov models (HMMs) to represent text [6,46]. Bikel et al.'s HMM has hidden states for each class of named entities and the not-a-name class. Each name class is generated conditioned on the previous name class, and uses learned bigram probabilities to generate a word. The Viterbi algorithm determines the most likely sequence of name class states given the sequence of words [6]. HMM-based methods do not require dictionaries or gazetteers, but are likely to make mistakes on words that did not appear in the training corpus.

The Stanford NER system [36] used by GeoCoder creates another type of hidden state sequence model, the Conditional Random Field (CRF), during its training. Unlike an HMM, a CRF can have state transition probabilities that vary depending on the position in the sequence and the input sentence. CRFs also allow the bi-directional flow of probabilistic information (so information at the end of the sequence can influence the beginning). The Viterbi algorithm is used to infer the most likely state sequence for both HMMs and CRFs. The Stanford NER algorithm also incorporates Gibbs sampling, which allows the CRF to model non-local structure. For example, given the input "the news agency Tanjug reported ... airport, Tanjug said ...", an NER system based solely on local information will classify the first Tanjug as an organization and the second as a person. A system that uses Gibbs sampling can enforce consistency, so that both are classified as organizations [12].

### 2.1.2. Recognizing Relationships

It is possible to use machine learning techniques to recognize the relationships between named entities in addition to the named entities themselves. For example, Wick et al. construct a fully-connected weighted graph, in which each vertex represents a field with a named entity and its type. Edge weights are determined by a compatibility function that represents the probability that two fields belong to the same record in a database. For example, an edge weight could represent the probability that a name and a phone number on a web page belong to the same person. The compatibility function is learned from labeled training data, and greedy agglomerative clustering is used to divide the graph into disjoint records [44]. Roth and Yih model relations, entities, and mutual dependences in a probabilistic framework. Separate classifiers are trained for entities and relations, and used to determine a conditional distribution for each entity and relation given the data. This information is combined with a belief network that represents constraints, and used to compute the most probable

class labels and relations. The system is capable of recognizing binary relationships, such as (born_in, person, location) [34]. This is useful for relationships that are described with combinations of verbs and prepositions.

### 2.1.3. Grounding and Disambiguating Locations

Once the location entities in a text are identified, each location name is grounded to a set of latitude and longitude coordinates. This involves disambiguating between all the entries in a gazetteer or GIS that match a location name. For example, GeoNames [1] can return latitudes and longitudes for all locations named "Boston" and a disambiguation system can determine that Boston, Massachusetts has latitude 42° 21' 30" and longitude 71° 3' 35". Disambiguation systems typically combine several heuristics to determine the location that a name is most likely to refer to. Heuristics are based on local context, discourse-level context, or location information. Leidner lists 15 common disambiguation heuristics [20]. For example, a city name followed by a qualifying state name is a common local context pattern. This information leads to accurate disambiguation, but won't be available for many place names. The *one referent per discourse* heuristic says that a location name will refer to the same location every time it's used in the discourse. The *geometric minimality* heuristic says that locations mentioned in a discourse will be as close together as possible. Finally, some heuristics use largest population or area to determine the most likely locations.

### 2.1.4. Representing Spatial Knowledge

When a grounded location name is part of a relative spatial reference, the remaining words in the reference need to be interpreted using contextual knowledge provided by a knowledge base. Given "south of Boston," the knowledge base should be able to list possible meanings corresponding to "south of." In this work the knowledge base is implemented through an ontology, which specifies the meanings of spatial words in terms of qualitative spatial relationships. For example, an ontology might include two individuals "Cambridge" and "Watertown", both instances of the *City* class and linked by the relationship *adjoining*. The Web Ontology Language (OWL) [23] is a common standard for encoding ontologies, and is used in GeoCoder.

Ontologies provide a representation of knowledge that is separate from algorithms and code, so that knowledge can be more easily extended. They have a hierarchical structure that approximates many domain models more compactly than algorithms or code. The use of ontologies does not rule out the possibility of developing a system that automatically learns its spatial knowledge. New knowledge could be used to augment or modify ontologies, and the system could continue to use them in later processing.

The most common formal representations of qualitative spatial relationships are region connection calculus (RCC-8) [32] and the 9-intersection model [10]. RCC-8 specifies eight base relations between two regions (shown in Figure 2.1): disconnected (DC), externally connected (EC), partially overlapping (PO), tangential proper part (TPP), inverse tangential proper part (TPPI), non-tangential proper part (NTPP), inverse non-tangential proper part (NTTPI), and equal (EQ).

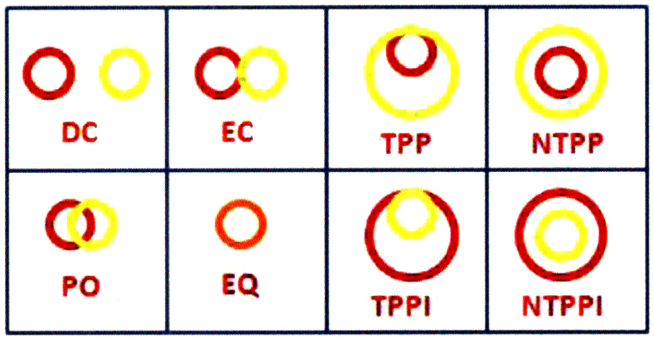


**Figure 2.1. RCC-8 Topological Relationships**

RCC-8 is complete, because it specifies every possible relation, and sound, because it only specifies possible relations. However, RCC-8 is limited because it can't express ambiguity. For example RCC-8 can't summarize a relation that could be tangential proper part or non-tangential proper part. Also, RCC-8 is incapable of expressing the difference between a region inside the hole of another region and a region outside the exterior boundary of the other region. Figure 2.2 illustrates this example.

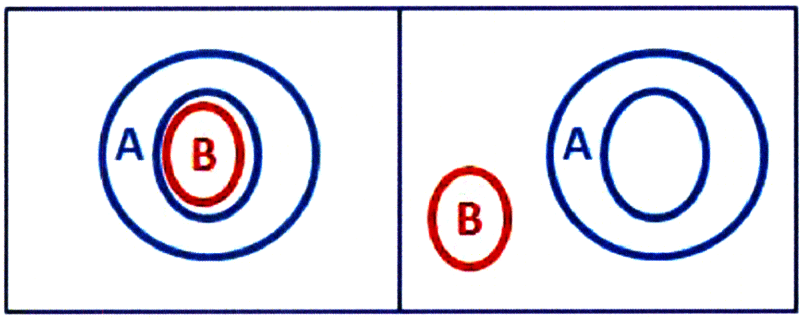


**Figure 2.2. Distinct Relationships that are Equivalent in RCC-8**

The 9-intersection model describes a spatial relationship through a 3x3 intersection matrix. Each region is decomposed into three parts. The rows correspond to the interior, boundary, and exterior of the first region, and the columns correspond to the interior, boundary, and exterior of the second region. If the set intersection between a component of the first region and a component of the second

region is non-empty, the corresponding element in the intersection matrix will be 1. Otherwise, it will be 0. Figure 2.3 gives an example of a spatial relationship and its intersection matrix.

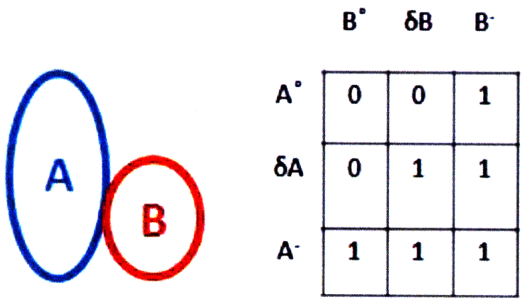


**Figure 2.3. A Spatial Relationship and Its Intersection Matrix**

This representation is complete but not sound, because many intersection matrices represent impossible relations. For example, it is impossible for a component of one region to intersect both the interior and exterior of another region, but not its boundary. This corresponds to an intersection matrix with [1 0 1] as a row. However, the 9-intersection model can handle arbitrary regions as well as points and lines.

El-Geresy and Abdelmoty introduce a generalized intersection model that can handle arbitrary decompositions of regions. They also specify rules that constrain the intersection matrices based on the underlying model of space. For example, the decomposition of each region completely covers the underlying space. Therefore, each component of the first region's decomposition must intersect at least one component of the second region's decomposition, since the two regions are in the same space. This rules out any intersection matrices with a column or row of zeros [11].

These representations of spatial relations rely on representations of space in which regions have well-defined boundaries, but a good representation for this ontology needs to handle the vagueness

common in spatial language. The spatial preposition "near X" does not specify a precise region, but a gradient that decreases with distance from X. Both RCC-8 and the 9-intersection model have been extended to support vague region boundaries. In fuzzy set theory [45], a region is characterized by a membership function, which assigns each point a grade of membership between 0 and 1. Union, intersection, and negation operations can be defined. In the "egg-yolk" model [7], a vague region is represented as a pair of concentric regions with defined boundaries. The "yolk" is small enough that it is almost certainly part of the vague region, and the "egg" is large enough that it almost certainly contains the entire vague region. The RCC relationships are defined in terms of the new regions. Both fuzzy set theory and the egg-yolk model create precise representations of vague regions. They fail to preserve information about the vagueness, and have no way to determine new representations if theirs turn out to be wrong. Anchoring theory [14] uses anchoring relations that express exactly what is known about the topological relationship between an object and a location, without requiring a precise mathematical representation like a fuzzy set. For example, an object is anchored outside a region if we are certain that no part of the object is located in the region. Anchoring relations are similar to RCC-8 relations, but the object is not a precise spatial location. There are some types of vagueness that can't be represented through an anchoring relation. For example, there is no way to anchor an object that touches the boundary of a region but could be inside or outside, or could overlap with the region.

### 2.1.5. Reasoning

Ontologies express axioms that can be used in automated logical reasoning. A sub-language of OWL (OWL-DL) is specifically intended for use with Description Logic (DL) reasoners. A Description Logic [5] is a logical formalism for expressing knowledge, which allows unary and binary predicates and restricted forms of quantification. A DL knowledge base consists of TBox axioms, which represent the

terminology of the application domain, and ABox axioms, which represent assertions. TBox axioms are class definitions and relationships between classes (e.g. "Farmers are people"). ABox axioms are assertions about specific individuals (e.g. "Bob is a farmer"). A DL reasoner performs three basic types of inference. In *subsumption reasoning*, a reasoner determines whether one concept is considered more general than another (e.g. the "city" concept is subsumed by the "populated place" concept). A reasoner checks *concept satisfiability* by determining whether a concept can have any instances (e.g. a concept defined as the union of "mountain" and "river" would not be satisfiable). Satisfiability is a special case of subsumption: if a concept is subsumed by the empty concept, then it is not satisfiable. Finally, a reasoner can perform *consistency checking* by determining whether the ABox has contradictions with respect to the TBox (e.g. an individual is an instance of the classes *Apple* and *Orange*, even though the TBox says that those classes are disjoint). Many tasks that involve reasoning about specific individuals can be reduced to consistency checking. All three types of inference are performed using tableau algorithms [5]. A tableau algorithm checks consistency by searching for a set of individuals that satisfies all the constraints. Description Logics vary in the types of axioms they can express. The simplest expressivity class, $\mathcal{AL}$ (Attributive Language), supports concept names and role names, concept intersection, universal restrictions, and limited existential number restrictions. OWL-DL supports the $\mathcal{SHOIN}$ expressivity class. The use of a $\mathcal{SHOIN}$ reasoner does impose some restrictions on the spatial knowledge in GeoCoder's ontologies. Specifically, it is not possible to express unions and intersections of roles (e.g. *intersects* and *touching*, *within* or *overlapping*), or qualified number restrictions (*intersects* exactly 3 *Cities*). However, $\mathcal{SHOIN}$ expressivity has been sufficient for GeoCoder's spatial knowledge, and unsupported axioms can be expressed through rules if necessary. Also, reasoning in $\mathcal{SHOIN}$ is known to be *decidable*. A more expressive class is $\mathcal{SROIQ}$, but it is not yet supported by OWL-DL. Each script letter represents a different extension to $\mathcal{AL}$ [47], and some common extensions are summarized in Table 2.1.

| Letter | Extension to $\mathcal{AL}$ |
|---|---|
| $\mathcal{C}$ | Complements |
| $\mathcal{S}$ | $\mathcal{C}$, and role transitivity |
| $\mathcal{H}$ | Role hierarchy |
| $\mathcal{R}$ | Complex Role Inclusions |
| $\mathcal{N}$ | Unqualified Number Restrictions |
| $\mathcal{Q}$ | Qualified Number Restrictions |
| $\mathcal{O}$ | Nominals |
| $\mathcal{F}$ | Functionality |
| $\mathcal{I}$ | Role Inverses |

**Table 2.1. Common Description Logic Expressivity Extensions**

A set of Description Logic axioms is *decidable* if there exists an algorithm that can check consistency in a finite amount of time. Recent research on DLs has focused on increasing expressivity while preserving decidability. Adding arbitrary rules to an ontology can make it undecidable, but adding DL-safe rules [25] preserves decidability. Each variable in a DL-safe rule is bound to an individual explicitly introduced in the ABox, so DL-safe rules can only be applied if the identities of all individuals are known.

In practice, qualitative spatial reasoning is intractable for a standard DL reasoner, because the ABox becomes too large [43]. The complete ABox contains a class assertion for each geographic feature, and a relation for each pair of features. Thus the size of the ABox is $O(n^2)$ in the number of features. Even with a small set of features, qualitative spatial reasoning can be intractable. One approach uses a table of the sets of possible compositions of pairs of spatial relationships in the TBox. For example, the composition of a tangential proper part relationship and a disconnected relationship is always a disconnected relationship. DL reasoners don't have the ability to simply look up an entry in the table. They must check the consistency of the entire composition table, and the proofs can require many

theorems and ad-hoc lemmas [31]. El-Geresy and Abdelmoty describe a method for computing the possible compositions of two intersection matrices by propagating empty and non-empty intersections. Based on the intersection matrix for regions A and B and the intersection matrix for regions B and C, they use a series of constraints to determine which components of the intersection matrix for regions A and C must be empty or non-empty [11].

Since a standard DL reasoner cannot scale for performing spatial reasoning, Wessel proposes using a hybrid representation and query language. This allows for a smaller ABox, because the system computes required relationships on the fly instead of computing all relationships before starting its reasoning. The system takes a "hybrid conjunctive query" and deduces the relationships that should be represented in the ABox. Wessel shows that query satisfiability and containment are decidable [43]. Hart and Dolbear suggest separating queries into spatial and non-spatial components. The spatial component is executed first, and the results are presented to the reasoner, which executes the non-spatial component [16]. This approach is limited because the non-spatial component has no way to execute additional spatial queries if it needs more information. The SPIRIT system [13] uses a domain ontology and a geo-ontology to resolve non-spatial and spatial terms, and iteratively expands the "footprint" of a query until it finds the desired result. For example, when looking for "castles near Edinburgh", it starts by looking within a short distance from Edinburgh. If it doesn't find any castles, it looks within a slightly larger distance, and continues to iteratively expand the search until it finds enough results. In summary, spatial and non-spatial queries can be executed separately, with different reasoners and ontologies, but an effective spatial reasoner needs to combine the results from spatial and non-spatial queries, and execute additional queries if necessary.

## 2.2. Related Work

The Related Work section discusses systems with goals that are similar to GeoCoder's, and approaches that GeoCoder attempts to improve upon.

### 2.2.1. Disambiguation Systems

There are several systems that use heuristics to disambiguate location names [3,21,22,33]. Rauch et al.'s system also computes the confidence of its judgments. The initial confidence is based on the average confidence of references in the training corpus. Local context changes the confidence that a reference is actually a location name (e.g. a location name preceded by "city of" has higher confidence that one preceded by "Mr."). Additional geographic references can increase the confidence of a disambiguation based on the *geometric minimality* heuristic. Other information, such as population, can also change the confidence estimates [33].

Disambiguation systems tend to be opaque to end users. A combination of heuristics is hard-coded, and can't be modified without a thorough understanding of the system. This is problematic because the best heuristics are likely to vary between applications. The heuristics that work best on a developer's test corpus may differ from those that would work best for another user's application. GeoCoder explicitly expresses heuristics in a knowledge base, enabling end users to easily modify heuristics or add new ones, and to thereby test various combinations of heuristics to find the one with the best performance. This also makes it possible to automatically select heuristics based on information about the text.

### 2.2.2. GeoLogica

SRI International's GeoLogica [42] is a system that automatically interprets geospatial questions. For example, GeoLogica can find a latitude/longitude pair for the Makuku Fossil Forest given the question "Show a petrified forest in Zimbabwe that is north of the capital of Botswana and within 200 miles of Lusaka, Zambia". Questions are parsed and translated into a logical form, which is passed to a reasoner. Its parser (Gemini) and reasoner (SNARK) were also developed at SRI.

Gemini is only used to parse questions, using a grammar and lexicon that were compiled from earlier projects, WordNet, and gazetteer sources. While it is capable of adding new words to the lexicon, it won't parse sentence structures that don't conform to its grammar. The grammar and lexicon are compiled by humans rather than learned from training data, so there is no way to train the system to parse sentences in other languages. Finally, Gemini doesn't handle word-sense ambiguity. Words like "north" and "in" correspond to single logical forms. In contrast, GeoCoder uses a parser that can be trained on new data, and is not limited to questions. It interprets simple patterns (e.g. a noun phrase containing a noun phrase and a prepositional phrase), so that part of a sentence can be interpreted even if the full sentence is not understood.

SNARK is an automated deduction system with the ability to obtain data from external knowledge sources through procedural attachment. SNARK has symbols that stand for external knowledge sources. When a proof uses an axiom containing one of these symbols, SNARK queries the external source. SNARK's external knowledge sources include the Alexandria Digital Library Gazetteer and the CIA World Factbook. GeoCoder's approach is similar, but queries to an external database are executed outside the reasoner, after it has inferred relationships between geospatial entities. This allows GeoCoder to use an existing reasoner without modifying its implementation. The reasoner can be updated or replaced without affecting other parts of GeoCoder.

### 2.2.3. Ordnance Survey Ontologies

Researchers at Ordnance Survey (the national mapping agency of Great Britain) have investigated ways to represent domain knowledge, database tables, and heuristics in OWL ontologies [9]. Their ontologies map domain concepts to a database, and can be used to generate SQL queries. They also developed a "nearness" heuristic based on Euclidean distance and gravity, which is high if an object is part of a cluster. This work overlaps with GeoCoder. GeoCoder uses ontologies to translate natural language into general queries, but does not yet have a model of SQL syntax in the ontology to facilitate automatic query generation. The Ordnance Survey ontologies translate the concept of "near" and a specific location into a SQL query, but do not interpret natural language or disambiguate place names or preposition senses.

# Chapter 3

# Spatial Knowledge Representation

GeoCoder uses a set of OWL-DL ontologies to represent and reason about spatial entities and relationships. OWL-DL ontologies describe classes and properties that are used in logical predicates. Classes (also called RDF types) are used in unary predicates, and properties are used in binary predicates. For example, a unary predicate might state that the individual "Boston" is an instance of the class *City*, and a binary predicate might state that the individual "Massachusetts" has a *capital* property with the individual "Boston" as its value. In the reasoner's internal model and in Jena's rule syntax, unary predicates are represented as binary predicates that use the *rdf:type* property. Predicates are written as RDF triples, for example:

("Boston" *rdf:type City*)

("Massachusetts" *rdf:type State*)

("Massachusetts" *capital* "Boston")

By convention, class names begin with an uppercase letter and property names begin with a lowercase letter. Property names sometimes begin with *has*, but "has" is an implied part of the meaning even if it is not part of the name. In this thesis, italics indicate class and property names.

The base ontology specifies the basic concepts used by the other ontologies. The relation ontology represents spatial relationships, including topology, direction, and distance. The domain ontology classifies types of spatial features. The parse ontology is used to translate a parse tree from the Stanford Parser into the format of the relation ontology. Finally, the query ontology is used to translate the ontology information into a set of spatial database queries or spatial functions that can be used to ground a feature. Table 3.1 lists the ontologies.

| Ontology | Purpose | Example Class or Property |
|---|---|---|
| Base Ontology | Define basic concepts shared between ontologies | *Feature* |
| Relation Ontology | Define a formal set of spatial relationships and selected spatial prepositions | *properPart* |
| Domain Ontology | Define types of features in a particular domain | *City* |
| Parse Ontology and Rules | Translate a parse tree into ontology representation | *IN* (preposition tag) |
| Query Ontology | Translate features in the ontology into GIS lookup and computational geometry queries | *intersects* |

**Table 3.1. List of ontologies used by GeoCoder**

The ontologies are separated so that there are as few dependencies as possible between them. This way, an individual ontology can be replaced without requiring any changes to the others. For example, different domain ontologies could be used for texts dealing with air and water, or in different languages, or a new parse ontology could be developed to interpret a tree with different part-of-speech tags (e.g. "P" instead of "IN" for preposition). Some dependencies are necessary, because the reasoner makes

inferences from an ontology to the other ontologies it depends on. Figure 3.1 shows the dependency structure of the ontologies (which is also the direction of inference).

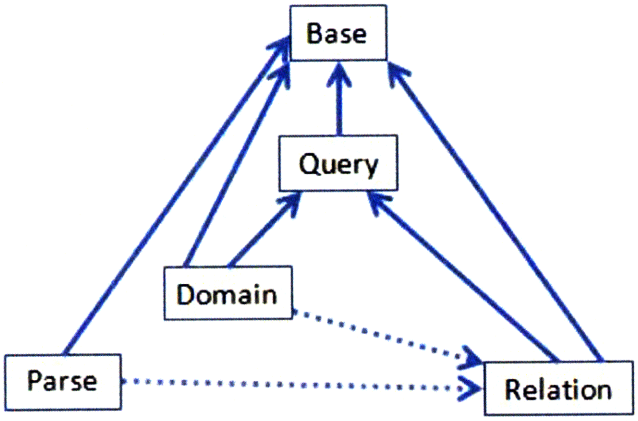


**Figure 3.1. Dependencies between the Ontologies**

Arrows indicate dependency, and dotted arrows indicate dependency through rules. All the other ontologies depend on the base ontology, the relation ontology and the domain ontology depend on the query ontology, and the domain ontology and the parse ontology depend on the relation ontology through rules.

## 3.1. Base Ontology

The spatial ontologies need to share basic concepts for a reasoner to make inferences between them. It has two classes, *Feature* and *FeatureSet*. The *FeatureSet* class is used to represent groups of one or more *Feature*s. For example, in the phrase "between Boston and New York", "Boston and New York" is represented as a *FeatureSet* with two features, "Boston" and "New York". A *FeatureSet* is related to its *Feature*s through assertions using the *hasFeature* property, which is also included in the base ontology. The "Boston and New York" example requires five assertions:

("Boston and New York" *rdf:type FeatureSet*)

("Boston" *rdf:type Feature*)

("New York" *rdf:type Feature*)

("Boston and New York" *hasFeature* "Boston")

("Boston and New York" *hasFeature* "New York")

Finally, the base ontology defines the *relation* property, which represents relationships. The *relation* property can be used with *Feature*s or *FeatureSet*s. Because OWL-DL properties are always binary, a ternary relationship like "between" is represented as a *relation* between a *Feature* and a *FeatureSet*. For example, "the city between Boston and New York" is represented as:

("Boston and New York" *rdf:type FeatureSet*)

("the city" *rdf:type Feature*)

("the city" *between* "Boston and New York")

There are multiple ways to represent spatial relationships in an ontology. This approach represents relationships as properties, but another possibility is to represent spatial relationships as classes instead. This representation seems counterintuitive and requires extra assertions, but it has advantages for some complex relations. For example, the "between" relation has at least two objects. When relations are represented as classes, the "between" class can allow multiple predicates with a *hasObject* property, and it isn't necessary to group *Feature*s. In this representation, "the city between Boston and New York" is:

("Boston" *rdf:type Feature*)

("New York" *rdf:type Feature*)

("between" *rdf:type Between*)

("between" *hasObject* "Boston")

("between" *hasObject* "New York")

("between" *hasSubject* "the city")

Additionally, OWL-DL reasoners allow more complex operations (including negation and intersection) on classes than on properties. For example, in the property representation there is no way to state that some relation is equivalent to (*touches* and not *intersects*).

While the class-based representation was considered, the property-based representation was ultimately chosen for its greater simplicity and clarity. This required the introduction of the *FeatureSet* class, and prevented the use of complex operations to relate the relation ontology to the query ontology.

## 3.2. Relation Ontology

The relation ontology provides a formal representation for topology relations, direction relations, and distance relations, as well as some natural language relations. The topology relations are represented in the RCC format [32], including both RCC-8 relations and RCC-5 relations. Figure 3.2 shows the RCC-8 and RCC-5 relations.

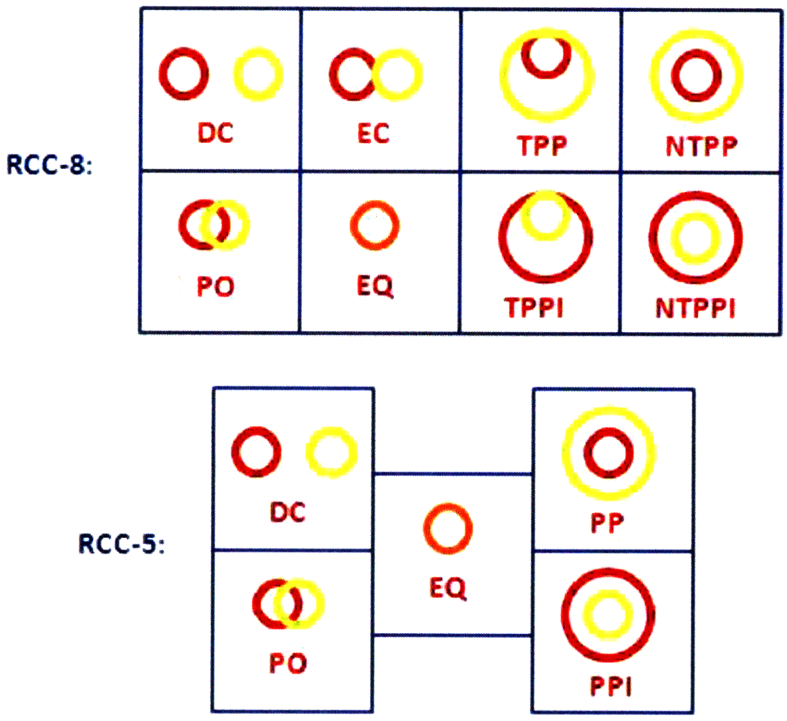


**Figure 3.2. RCC-8 and RCC-5 Relations**

RCC-8 is sometimes too specific (e.g. natural language statements usually do not distinguish tangential proper part relations from non-tangential proper part relations), so including the RCC-5 relations for proper part (PP) and inverse proper part (PPI) allows the ontologies to preserve some ambiguity when translating relations into spatial queries. An intersection model [10] decomposes regions into parts (e.g. interior, boundary, and exterior), and lists the intersections between parts in a matrix (see Figure 2.3). This representation would allow for more ambiguity and partial information, but would require the ontology to represent many more relations, including many that could never actually occur (such as a matrix with a row of zeros). RCC was chosen over the intersection model representation because it only includes a small set of possible relations, and can be completely represented through properties in the ontology. Each of the RCC relations is represented as a property, with appropriate subproperty assertions (e.g. *tangentialProperPart* is a subproperty of *properPart*).

The eight cardinal directions are represented as disjoint properties: *north*, *northeast*, *east*, *southeast*, *south*, *southwest*, *west*, and *northwest*. The ontology could easily be extended to include more directions, such as east-southeast, but the additional directions are much rarer in natural language. Relations between directions (e.g. northeast is between north and east) are also ignored in the current version of the ontology. The distance relation properties are *near* and *far*, and their meanings can be interpreted based on information in the domain ontology. Each of the topology, direction, and distance properties includes subproperty or equivalent property assertions that relate it to one or more properties in the query ontology. For example, *partiallyOverlapping* has the equivalent property *overlaps*, which is a property in the query ontology. Table 3.2 lists some representative properties and their superproperties from the query ontology.

| Property | Type of Property | Superproperties from Query Ontology |
|---|---|---|
| *externallyConnected* | Topology relation | *touches, intersects* |
| *properPart* | Topology relation | *intersects, within* |
| *North* | Direction relation | *north* |
| *Near* | Distance relation | *withinDistance* |

**Table 3.2. Properties in the relation ontology**

Finally, the relation ontology includes properties for a set of spatial prepositions (currently *of*, *in*, *near*, and *around*) with subproperties to represent the different senses of the preposition. For example, the preposition "around" has senses that mean "outside of", "near", and "throughout" (according to Merriam-Webster's online dictionary [24]). Each sense also has a topology or distance relation as a superproperty. For example, the "outside of" sense of "around" has the *properPartInverse* property as a superproperty. The "near" sense has *near* as a superproperty, and the "throughout" sense has *equal* as a superproperty. The prepositions and senses can also specify different domains and ranges based on

classes in the domain ontology. For example, the object of "around" could be an instance of the *City* class but probably not the *River* class, whereas the object of "along" could be a *River* but not a *City*.

## 3.3. Domain Ontology

The domain ontology extends the *Feature* class from the base ontology. This ontology is meant to be extended, modified, or replaced for specific applications with their own vocabularies. A domain ontology can contain as much detail as necessary. The *Feature* class is divided into the subclasses *NamedFeature* and *UnnamedFeature*. The *NamedFeature* class is used for phrases that are recognized as location names, and can be looked up in the GIS. The *UnnamedFeature* class is used for words or phrases that refer to spatial entities but are not names, such as "the city" or "the north". The *UnnamedFeature* class is subdivided into two classes: *ExistingFeature* and *DerivedFeature*. When designing a new domain ontology, a person specifies words that refer to entities in each class. An *ExistingFeature* is an entity which corresponds to an entry in the GIS. Because its name is unknown, the query to the GIS must be created from the entity's relations to other entities' geometries. For example, in the phrase "the city near Boston and Cambridge", "city" is considered an *ExistingFeature*, because the word "city" corresponds to a type of feature in the GIS. Some of the subclasses of *ExistingFeature* included in the sample domain ontology are *City*, *Country*, *Lake*, *Mountain*, *River*, and *Road*. These subclasses could be extended or modified based on the capabilities of the GIS and the types of words that are likely to occur in other applications. The subclasses could also be arranged into a hierarchy: for example, a class for bodies of water could include *River* and *Lake* as subclasses. A *DerivedFeature* represents a component of another entity's geometry, and therefore does not itself exist in the GIS. For example, in the phrase "in the west of Boston", "the west" is considered a *DerivedFeature*. Each subclass of *DerivedFeature* corresponds to a class in the query ontology, which names a computational geometry

function that can be performed to obtain the geometry of the *DerivedFeature*. In the current version of the domain ontology, the only subclasses of *DerivedFeature* are the four directions *North*, *South*, *East* and *West*. Other possible subclasses might include *Center*, *Border*, *Area*, or *Metropolis*. Finally, the domain ontology includes *GroundedFeature* and *GroundedFeatureSet* classes. These classes are used to mark *Feature*s and *FeatureSet*s as grounded. One of these classes is added to a *Feature* or *FeatureSet* individual once GeoCoder has found a set of possible geometries for it.

## 3.4. Parse Ontology and Rules

The Stanford Parser converts a phrase into a tree with Penn Treebank [28] tags. After the tree is created, GeoCoder translates it into the parse ontology by recursively creating individuals for every subtree. Each individual has a class corresponding to the Penn Treebank tag of its root. The tree's leaves (terminals) contain the words, which are mapped to classes in the domain ontology or the relation ontology when they match the name of a class. Words which have been recognized as location names are instances of the *NamedFeature* class. Nodes above the leaves (pre-terminals) contain the words' part-of-speech tags. The parse ontology also specifies a single property, *child*, which preserves the structure of the tree. The translation algorithm is described in Section 4.3, and pseudocode is included in Appendix A.1.

The parse ontology also relies on a separate file of rules in Jena rule format [8]. A rule consists of a body and head, each of which consists of one or more triples. The triples can contain variables, which are shared among all the triples in a rule, but not across different rules. A Jena rule reasoner tries to match individuals in the ontology to the triples in a rule's body. If there is an assignment of variables that makes all the triples in the rule body match triples in the ontology model, the triples in the rule

head are added to the ontology. The Jena rule reasoner continues to run until it cannot add any additional triples to the ontology. For example, a rule in Jena format is:

[ (?x *child* ?y), (?x *rdf:type Woman*) -> (?x *mother* ?y) ]

If the ontology contains an instance of the *Woman* class and a property assertion that relates that instance to another individual through the *child* property, a new property assertion will be added to relate the two individuals through the *mother* property.

The rules are used to translate tree structure patterns into predicates using the properties in the relation ontology. The first rules are used to recognize patterns that should become *FeatureSets*. Any instance of the *NP* class with an instance of the *Feature* class as a child becomes a *FeatureSet*, with *hasFeature* property assertions relating it to its child *Features*. These rules are:

[ (?a *rdf:type* parse:*NP*), (?a parse:*child* ?b), (?b *rdf:type* base:*Feature*) -> (?a *rdf:type* base:*FeatureSet*) ]

[ (?a *rdf:type* base:*FeatureSet*), (?a parse:*child* ?b), (?b *rdf:type* base:*Feature*) -> (?a base:*hasFeature* ?b) ]

The rest of the rules recognize mid-level patterns containing a noun phrase with a prepositional phrase as a modifier. They don't depend on high-level sentence structure or low-level part-of-speech structure. These rules translate the preposition into a relation ontology property, with the noun from the noun phrase as one object and the *Feature* or *FeatureSet* in the prepositional phrase as the other. Their rule bodies include class assertions about nodes in the tree as well as *child* property relationships between nodes in the tree. An example is:

[ (?a parse:*child* ?b), (?a *rdf:type* parse:*NP*), (?b *rdf:type* parse:*NP*), (?a parse:*child* ?c), (?c *rdf:type* parse:*PP*), (?c parse:*child* ?d), (?c parse:*child* ?e), (?e *rdf:type* parse:*NP*), (?d *rdf:type* domain:*in*), (?b parse:*child* ?g), (?g *rdf:type* parse:*Noun*), (?e parse:*child* ?f), (?f *rdf:type* base:*FeatureSet*) -> (?g relation:*in* ?f) ]

## 3.5. Query Ontology

The query ontology provides a list of functions that can be used to ground instances of the *UnnamedFeature* class. *ExistingFeature*s are looked up in a GIS using PostGIS [2] functions, while *DerivedFeature*s are computed using computational geometry functions, implemented using the Java Topology Suite (JTS) [40].

PostGIS queries the GIS for an entity based on its relations to other geometries. PostGIS queries are represented as properties, with the desired *Feature* as one object and the reference *Feature* or *FeatureSet* as the other. For example, a triple with a PostGIS query is:

("state" *contains* "Boston and Cambridge")

The possible PostGIS queries are: *contains*, *disjoint*, *equals*, *intersects*, *overlaps*, *touches*, *within*, *withinDistance* (used with "near"), *intersects-buffer-convexHull* (buffers the convex hull of a set of features by a fraction of the area of the convex hull – used with "between"), and the cardinal directions. The query ontology also includes the *orderBy* property. Value restrictions on this property can be added to the domain ontology to specify the database attribute that should be used to order results. For example, instances of the *City* class could be ordered by population, and instances of the *Lake* class could be ordered by area. The *orderBy* property adds an "order by" clause to the end of the SQL query generated from the query ontology. Using the *orderBy* property helps to ensure that the most relevant results are displayed first.

Computational geometry functions are used to ground *DerivedFeature*s by performing defined spatial functions on the geometries of reference *Feature*s or *FeatureSet*s. The reference geometries are polygons that have been looked up in the GIS in earlier processing steps. The computational geometry functions are represented as classes, because the appropriate functions tend to depend on the noun

(e.g. "center") rather than the preposition (usually "of"). Nouns are translated into classes in the domain ontology, and can easily have superclasses in the query ontology. For example, in the phrase "the north of Boston", "north" is translated into the class assertion:

("north" *rdf:type North*)

The *North* class in the domain ontology is a subclass of *NorthTriangle* in the query ontology, so another class assertion is inferred:

("north" *rdf:type NorthTriangle*)

The *NorthTriangle* class specifies a computational geometry function that computes the northern part of a geometry by using two 45° lines and the geometry's northern boundary to define a triangle. However, the properties of a *DerivedFeature* are still computed and used to determine the reference *Feature*. In this case, the property assertion:

("north" *of* "Boston")

is computed, and "Boston" is the reference feature. The queries in the current query ontology are *NorthTriangle*, *SouthTriangle*, *EastTriangle*, and *WestTriangle*. Functions that perform each of these queries are implemented in Java, and more queries could be added in the future.

# Chapter 4

# GeoCoder Server

The GeoCoder Server combines named entity recognition, parsing, knowledge representation, reasoning, and spatial queries and functions into an integrated system that converts natural language phrases into GML strings that can be displayed with programs like Google Earth. GeoCoder recognizes location names, parses input phrases, and builds an ontology model with location references and their relationships. Relationships are determined using phrase-level structural patterns. GeoCoder grounds the location references iteratively, based on their relationships to previously grounded locations. This chapter describes each of these processing components, as well as the server's user interface and the process of communicating with clients.

## 4.1. Server User Interface and Client-Server Interface

Running the server program opens a single window with buttons for starting and stopping the server, and the ability to select the locations of the ontologies and the spatial database that the server should use. A screenshot of the server window is shown in Figure 4.1.

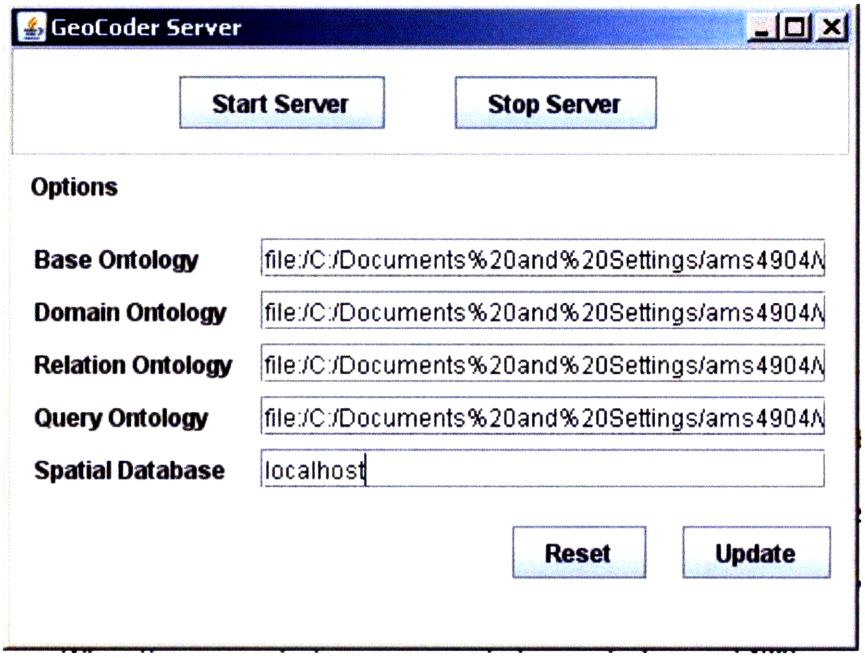


**Figure 4.1. Screenshot of the GeoCoder Server**

Though the interface doesn't allow the user to modify the locations of files related to the parser, including the parse ontology and rules and the grammar file, this ability could easily be added in the future.

When the server starts, a server socket is created on port 4206, and a new thread is launched. The thread initializes the named entity tagger and the parser, and waits for clients to connect. Whenever a client connects to the socket, the server starts a processor thread to communicate with that client. Sections 4.2 through 4.6 discuss the actions of a single processor thread on the server. The

server expects the input text followed by a line that starts with the "|" character. This line can contain a single "|", or any combination of the output options "|VERSION", "|TAG", and "|PARSE". The final output line is always an XML string followed by "|", but specifying one or more of the output options will cause the server to provide intermediate outputs. "|VERSION" gives the name and version number of the server, and can be used to check that the client's connection to the server. "|TAG" will return the input text with recognized entities marked as "/LOCATION", "/PERSON", or "/ORGANIZATION". The "|PARSE" option will return the parse tree with named entity tags included. These options are useful for checking if an unexpected output could be due to an error by the tagger or parser.

This client-server interface was chosen because it requires minimal processing on both the client and server sides. The "|" character is not used in natural language, so there is no possibility of confusing the input text and the options. The client and server could use a more standard format (e.g. XML) to communicate, but the client would have to do additional encoding of the input, and the server would have to decode it. Also, the server maintains a persistent connection to the client for as long as possible. This way, the server can use a single processor thread for multiple queries from the same client. If the connection is lost, the client can reconnect and the server will launch a new processor thread. In the future, the server could maintain some state between queries (e.g., to use a disambiguation heuristic that sorts by proximity to previous results). The server would need a better method for handling a dropped connection. For example, the server could keep the processor thread running until it received a termination signal from the client. The client could disconnect and reconnect without losing its processor thread.

## 4.2. Named Entity Tagging and Parsing

The first processing steps are tagging the named entities in the input text and parsing the resulting text into a tree. GeoCoder uses the named entity tagger and parser developed by the Stanford Natural Language Processing Group [36,37]. The tagger and parser are loaded once and shared among processor threads. Each is used by one thread at a time, to ensure thread safety. This can slow down the server if there are multiple clients, but the tagger and parser require too much memory to run multiple copies simultaneously.

The Stanford NE tagger tags each word that is recognized as part of a named entity with either "/LOCATION", "/PERSON", or "/ORGANIZATION". (It tags everything else with "/O", but these tags are removed by the server.) For example, the phrase

"the park east of Lake Quannapowitt"

is converted to

"the park east of Lake/LOCATION Quannapowitt/LOCATION"

If two or more words with "/LOCATION" tags are adjacent, the server combines them into one word:

"the park east of Lake_Quannapowitt/LOCATION".

The output from the tagger is passed to the Stanford Parser. This is a statistical parser trained on the Penn Treebank, so it uses Penn Treebank-style trees and tags. In addition, the parser's accuracy is limited by the training data. There are relatively few geospatial references in the Penn Treebank. An important example is direction words. In the sentence "The park is *direction-word* of Boston", where *direction-word* is e.g. "northwest", *direction-word* could get tagged as a noun, an adjective, and adverb, or a superlative adjective depending on whether the word is "south", "east", "northwest", or

"southwest". This thesis focuses on examples that the parser handles correctly. Performance could be improved by retraining the parser on a corpus with more direction word examples. The new corpus would have to be annotated with Penn Treebank-style parse trees by hand, so this improvement was not attempted here.

Later stages of processing assume that the parser was correct, and do not try to detect or correct errors. However, many sentences (particularly those with many prepositions) have ambiguous parses, and in some cases information from later stages could be used to disambiguate them. For example, "the park near the lake in Boston" has two possible parses:

(the park (near the lake) (in Boston))

and

(the park (near the lake (in Boston)))

In the first case, the park is in Boston, but the lake might not be. In the second case, the lake is in Boston, but the park might not be. At the parsing stage there is no way to determine which is correct. Later, however, we might find that there are no lakes in Boston, and be able to eliminate the second case. A future version of GeoCoder could store a set of possible parses and eliminate those that are found to be impossible as it grounds features. This will require a method of determining which parses are semantically different.

## 4.3. Ontology Representation and Reasoning

The server loads all the ontologies described in Chapter 3 into a Pellet reasoner [35] ontology model. The Pellet reasoner makes inferences based on information in the ontologies (e.g.

subclass/superclass assertions or restrictions on properties), and adds new assertions to the ontology model based on its inferences. The server then traverses the parse tree and converts every node into an individual, using the algorithm in Appendix A.1. It starts by creating an individual for the root. It also creates individuals for each of the root's children, and relates them to the root with the *child* property. Every individual is an instance of the *Node* class, and some noun individuals are given additional classes as follows. If a noun contains the "/LOCATION" tag, the individual will be an instance of the *NamedFeature* class. If the noun matches the name of a class in the domain ontology, the individual will be an instance of that class. The process recurses with each of the children as the root, until the entire tree is represented in the ontology model. The Pellet reasoner makes any inferences it can. For example, the reasoner could use the assertion

("city" *rdf:type City*)

and *City*'s subclass/superclass relationship to *ExistingFeature* to infer

("city" *rdf:type ExistingFeature*)

The server then copies the ontology model into a new ontology model with a Jena rule reasoner. The Jena reasoner uses rules from the parse rules file to convert the information in the parse ontology format into relation properties in the relation ontology format. This process was described in Section 3.4. While the Pellet reasoner could be used to apply rules to avoid copying the ontology model, Pellet's current rule implementation is impractically slow. The new assertions are added to the original ontology model (with the Pellet reasoner), and the Pellet reasoner then uses the relation properties to infer query properties in the query ontology format. For example, if the assertion

("city" *around* "Harwood Heights")

is added by the Jena reasoner, the Pellet reasoner will infer

("city" *properPartInverse* "Harwood Heights")

("city" *contains* "Harwood Heights")

based on subproperty assertions in the relation ontology.

## 4.4. Grounding

Once all the information is represented in the ontology model, the server attempts to ground the features using the algorithm in Appendix A.2. First, it finds all the instances of the *NamedFeature* class and looks up their names in the GIS. The GIS returns a set of one or more geometries corresponding to each name. If there are multiple geometries, they form an ambiguity set of candidate geometries. If, in later steps, one of the geometries is found to be inconsistent with other information, it is removed from the set. For example, the US Census Bureau's TIGER database [38] contains three geometries for the name "Boston", one in Massachusetts, one in Georgia, and one in Indiana. If the phrase is "the river in Boston" and we find that there is no river in Boston, Indiana, that geometry is removed from the set. If none of the geometries are inconsistent, they all remain in the set and are included in the final output.

Next, the server enters a loop in which it attempts to ground any ungrounded *Feature*s or *FeatureSet*s. *Feature*s and *FeatureSet*s are marked as grounded after a set of candidate geometries is found. (This is done by adding a class assertion with the *GroundedFeature* class or the *GroundedFeatureSet* class.) The server finds the set of ungrounded *NamedFeature*s and the set of ungrounded *FeatureSet*s. If both sets are empty or neither set has changed size since the last iteration, the server is finished and breaks out of the loop. Otherwise, the server attempts to ground each ungrounded *Feature* and *FeatureSet*, as follows.

First, the server attempts to ground the ungrounded *Features*. If an ungrounded feature is an *ExistingFeature*, the server searches the ontology model for query properties that relate the ungrounded feature to a grounded *FeatureSet*, such as

("city" *touches* "Boston")

The *touches* property indicates that the boundary of "city" touches the boundary of "Boston". Note that every *Feature* is a member of a *FeatureSet* (and many *FeatureSets* contain only one *Feature*). Therefore, an ungrounded feature like "city" can't necessarily be grounded once the "Boston" *Feature* has been grounded – the "Boston" *FeatureSet* must be grounded first. This means that the server's first attempt to ground ungrounded features will fail, because the reference *FeatureSets* have not been grounded yet. If there is a query property that relates the ungrounded feature to a grounded *FeatureSet*, the server uses a PostGIS query to find the set of geometries for the ungrounded feature, based on its relationship to the grounded *FeatureSet* (see Chapter 5 for examples). Otherwise, it tries the next grounded *FeatureSet*.

If an ungrounded feature is a *DerivedFeature*, the server looks for a *relation* property that relates the ungrounded feature to a grounded *FeatureSet*. If there is one, the server searches the ontology model for a query class assertion involving the ungrounded feature, such as

("south" *rdf:type SouthTriangle*)

and calls a computational geometry function (which uses JTS) on the reference *FeatureSet*'s geometry set to find a set of candidate geometries for the ungrounded feature. Each query class matches a computational geometry function implemented in the server. For example, if the ungrounded feature is an instance of the *SouthTriangle* query class, the server calls a function that computes the southern

triangle of the reference *FeatureSet*'s geometry. For both types of queries, the ungrounded feature is marked as grounded after its candidate geometry set has been found.

Next, the server attempts to ground the ungrounded *FeatureSet*s. For each set, it searches the ontology for all the *Feature*s that are related to the set through the *hasFeature* property. For example, the assertion

("Boston and New York" *hasFeature* "Boston")

indicates that "Boston" is related to the *FeatureSet* "Boston and New York" and should be included in the set of members of the *FeatureSet*. If all the member *Feature*s are already grounded, then a computational geometry operation can be used to create a set of candidate geometries corresponding to the *FeatureSet*, and the *FeatureSet* is marked as grounded.

Throughout this process, the sets of candidate geometries (in the "well-known text" (WKT) markup language [27]) corresponding to each *Feature* are stored in a hash map. Another option is to use the grounding loop to compose a long PostGIS query for the entire phrase, and only go to the GIS database once, when the query is complete. The database queries are the processing time bottleneck for most natural language phrases, so this method could greatly increase GeoCoder's speed. However, it would be difficult to accommodate multiple databases, because they cannot be queried at the same time. It would also be difficult to use separate tables for separate feature types, because the tables would have to be joined before they could be queried together. Also, this method decreases the transparency of the server, as GeoCoder wouldn't be able to return any intermediate geometries. In this early version, visibility is more important than speed, but future versions could improve performance by reducing the number of database queries.

## 4.5. GIS Queries

The server performs two types of GIS query: grounding a feature based on its name, and grounding a feature based on its relationship to a set of reference feature geometries. The GIS is a database that stores location entities. Each entity has a name and a WKT [27] geometry, which contains a list of coordinates that define the location's polygon. To ground a feature from its name, the server simply retrieves the geometries for entities with that name, by issuing a SQL query. It is also possible to specify an *orderBy* property for the *NamedFeature* class in the domain ontology to sort the results. Grounding based on a relationship is similar, but the SQL query's "where" clause uses PostGIS functions instead of the name. In both cases, the GeoCoder server traverses the GIS results and creates a set of candidate geometries.

## 4.6. Computational Geometry Functions

Computational geometry functions use the Java Topology Suite (JTS) to ground features by performing geometric functions on sets of reference features' candidate geometries, and to ground feature sets by combining the candidate geometries of the member features. To ground a feature based on a reference feature, the server performs an operation on each of the reference feature's candidate geometries separately, and returns the set of result geometries. The current functions are north, south, east, and west triangles. Each finds the centroid of a geometry, and creates a new geometry using 45° lines that extend from the centroid to the boundary of the original geometry. This isn't necessarily a correct interpretation of the direction words. It would be better to determine the interpretation based on the shape of the reference geometry. For example, many cities are bisected by rivers, which form a natural boundary between north and south or east and west. It would also be possible to create

multiple computational geometry functions for directional partitioning, and include them all in the query ontology. Users could specify the appropriate function to use for a certain type of feature in the domain ontology.

A *FeatureSet* is grounded using the candidate geometry sets of each of its member *Features*. Each of the *FeatureSet*'s candidate geometries is a JTS GeometryCollection, consisting of one candidate geometry from each member's candidate geometry set. A GeometryCollection is created for each combination of the members' candidate geometries. At this time, the GeometryCollections are not ordered, but it would be possible to order them, for example, based on the area of the union of the geometries or the area of the convex hull of the geometries.

## 4.7. GML Output and Display

After the server has grounded as many features as possible, it passes the hash map of features and candidate geometries to a function that converts the map into a XML string. A sample XML string is in Appendix A.3. The XML string contains an <ENTITY> tag for each candidate geometry, so there can be multiple alternative entity tags corresponding to a single feature. The <ENTITY> tags for a feature will be sorted, so the best geometry will correspond to the first tag. Each <ENTITY> tag contains a <NAME> tag, with the input phrase used to generate the geometry, and a <GML> tag, with a representation of the geometry in GML (as determined by JTS's GMLWriter [39]).

The first element of the GML geometry is either <gml:Polygon> or <gml:MultiPolygon>. In this thesis, Polygon and MultiPolygon are used with same meanings they have in GML, though the actual geometries may be represented in another format, such as WKT or KML. A MultiPolygon is used when the JTS geometry is a GeometryCollection. It consists of <gml:polygonMember> elements, each of which

contains a <gml:Polygon> or another <gml:MultiPolygon>. This allows a MultiPolygon to consist of multiple disjoint polygons. For example, the MultiPolygon for San Francisco includes a disjoint polygon for a set of islands (shown in Figure 4.2).

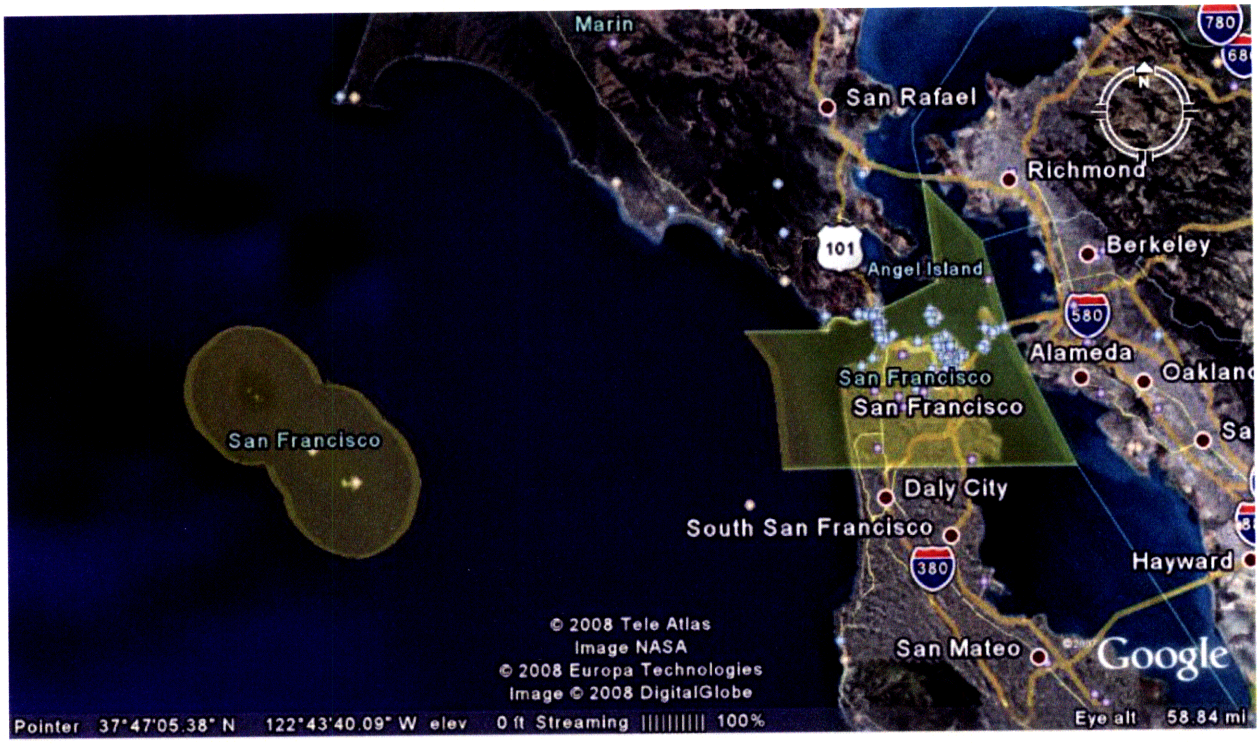


**Figure 4.2. MultiPolygon for San Francisco**

GeoCoder currently uses data from the US Census Bureau's TIGER database [38], in which cities are represented as MultiPolygons, some of which consist of only one Polygon. MultiPolygons are also used when separate members of a *FeatureSet* are combined into a GeometryCollection. The features' original geometries are preserved in the new MultiPolygon (even adjacent Polygons are not combined). For example, Figure 4.3 shows the MultiPolygon for "San Francisco and Redwood City". This MultiPolygon consists of one MultiPolygon (for San Francisco) and one Polygon (for Redwood City).


**Figure 4.3. MultiPolygon for "San Francisco and Redwood City"**

Each <gml:Polygon> element contains a <gml:outerBoundaryIs> element, which consists of a list of coordinates that defines the outer boundary of the polygon. A polygon can also have additional coordinate lists in one or more <gml:innerBoundaryIs> elements, if it contains holes. For example, the polygon of Oklahoma City (shown in Figure 4.4) contains multiple inner boundaries.

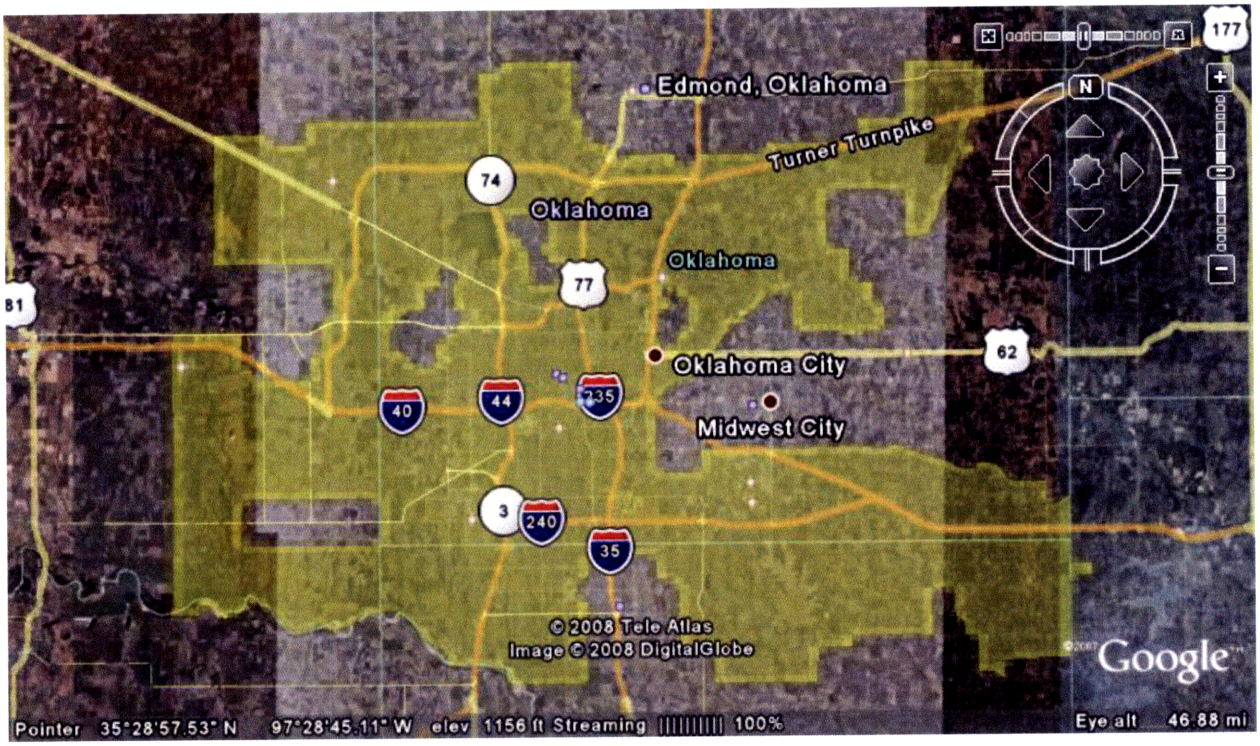


**Figure 4.4. Polygon for Oklahoma City, with multiple inner boundaries**

A GeoCoder client can extract information from the XML string for display. While the server could output a KML string (which can be directly displayed in Google Earth, for example), many aspects of the display (e.g. color and elevation to zoom to) would be fixed. Using the XML string gives the client more flexibility. My examples use a GeoCoder client developed by Erik Antelman at Draper Laboratory. The client transforms the XML string into KML, and adds the results to Google Earth's "Temporary Places" folder. Figure 4.5 shows a screenshot of this client, with the query "the city near the north of New York".

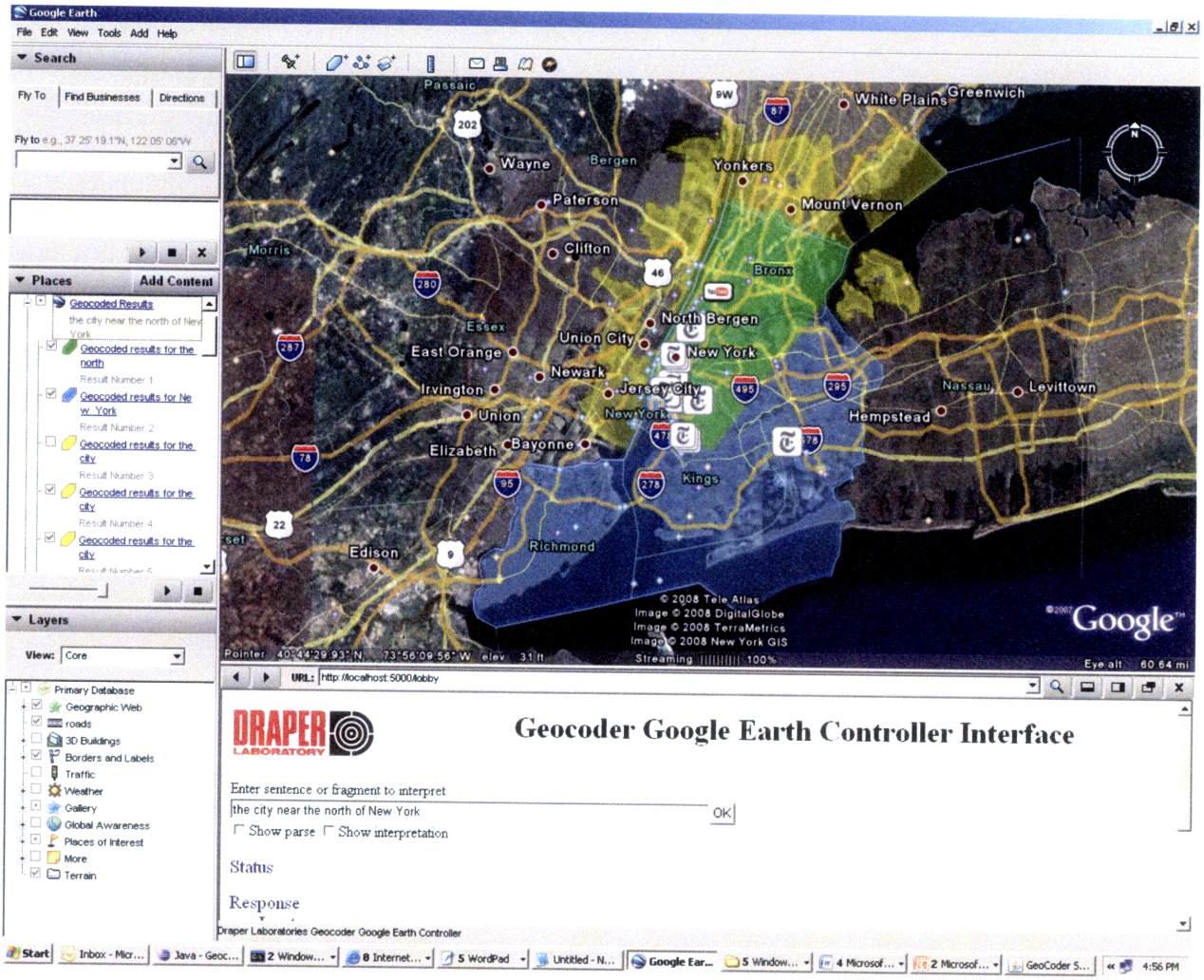


Figure 4.5. Screenshot of a GeoCoder Client

# Chapter 5

# Examples

This chapter works through a series of examples to demonstrate GeoCoder's processing. The processing steps are:

1. Named Entity Tagging
2. Parsing
3. Translation to Parse Ontology
4. Application of Rules
5. Ground the *NamedFeatures*
6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets*
7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures*

   Repeat steps 6 and 7 as necessary
8. Create an XML output string
9. Display

Section 5.1 explains all the steps in detail for a simple example, and the other sections explain selected steps that become more complicated when processing other examples.

## 5.1. Unambiguous PostGIS Lookup: "the city near West Wendover"

The first example was chosen because it is an unambiguous reference, based on data from the TIGER database. Table 5.1 (at the end of this section) summarizes GeoCoder's processing of "the city near West Wendover". West Wendover, NV is just west of the Nevada-Utah border, and Wendover, UT is just east of it. West Wendover and Wendover are adjacent, and the next closest city, Wells, is about 40 miles away. Figure 5.1 shows West Wendover, Wendover, and Wells on Google Maps.

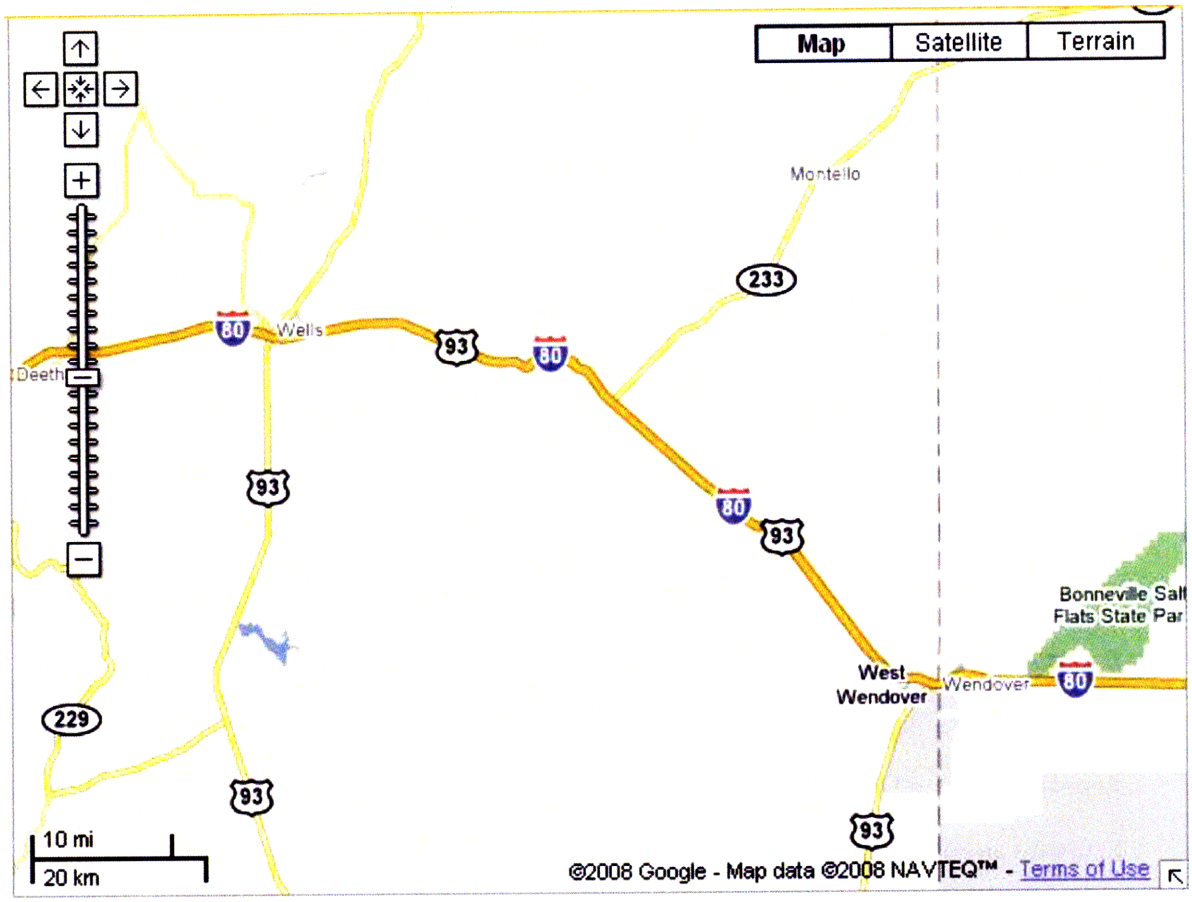


**Figure 5.1. West Wendover, NV on Google Maps**

*Step 1*: Named Entity Tagging. The server passes the input phrase "the city near West Wendover" to the named entity tagger. Result: "the city near West_Wendover/LOCATION".

*Step 2:* Parsing. The tagged phrase is passed to the parser, which returns the parse tree in Figure 5.2.

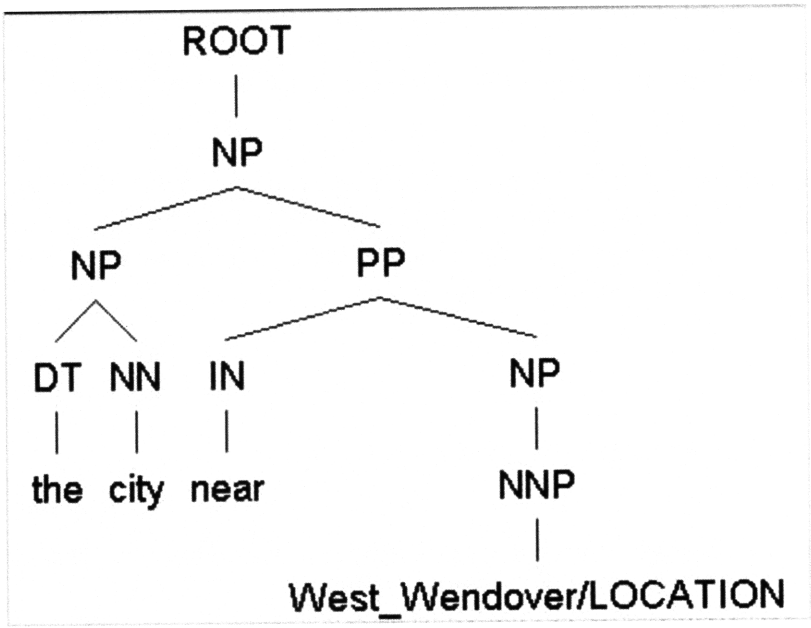


**Figure 5.2. Parse tree for "the city near West_Wendover/LOCATION"**

*Step 3:* Translation to Parse Ontology. The server traverses the tree and adds individuals corresponding to each node to the ontology model. The (NN city) node is given a class assertion of *City*, and the (NNP West_Wendover/LOCATION) node is given a class assertion of *NamedFeature*.

*Step 4*: Application of Rules. The server applies Jena rules from the rules file to recognize patterns in the tree structure. The (NNP West_Wendover/LOCATION) individual is an instance of the *Feature* class, so (NP (NNP West_Wendover/LOCATION)) matches the rule

> [ (?a *rdf:type* parse:*NP*), (?a parse:*child* ?b), (?b *rdf:type* base:*Feature*) -> (?a *rdf:type* base:*FeatureSet*) ]

This makes (NP (NNP West_Wendover/LOCATION)) a *FeatureSet*, and the rule

> [ (?a *rdf:type* base:*FeatureSet*), (?a parse:*child* ?b), (?b *rdf:type* base:*Feature*) -> (?a base:*hasFeature* ?b) ]

adds *hasFeature* (NNP West_Wendover). Similarly, (NP (DT the) (NN city)) becomes a *FeatureSet* with *hasFeature* (NN city). Also, the tree structure matches a rule for NP-PP trees:

> [ (?a parse:*child* ?b), (?a *rdf:type* parse:*NP*), (?b *rdf:type* parse:*NP*), (?a parse:*child* ?c), (?c *rdf:type* parse:*PP*), (?c parse:*child* ?d), (?c parse:*child* ?e), (?e *rdf:type* parse:*NP*), (?d *rdf:type* domain:*near*), (?b parse:*child* ?g), (?g *rdf:type* parse:*Noun*), (?e parse:*child* ?f), (?f *rdf:type* base:*FeatureSet*) -> (?g relation:*near* ?f) ]

This adds the assertion

> (NN city) *near* (NP (NNP West_Wendover/LOCATION))

The Pellet reasoner automatically infers the *withinDistance* property from the *near* property, because *withinDistance* is a superproperty of *near*. It also infers that (NN city) is an *ExistingFeature*, because *ExistingFeature* is a superclass of *City*.

*Step 5:* Ground the *NamedFeatures*. There is only one *NamedFeature*, "West Wendover". The server inserts the name into a SQL query:

> "select asText(the_geom) from place where name like 'West Wendover';"

The "asText" function computes the well-known text representation [27] of the geometry, and "place" is the name of the table containing the geometries. The query returns one result, a well-known text string corresponding to the geometry of West Wendover:

```
MULTIPOLYGON(((-114.112847 40.725489,-114.112591 40.726539,-114.112504
40.726894,-114.112751 40.733003,-114.112872 40.735993,-114.113024 40.739743,-
114.113021 40.749187,-114.112842 40.750482,-114.112852 40.750639,-114.112904
```

```
40.75146,-114.112828 40.751876,-114.112776 40.752163,-114.112627 40.752984,-
114.112976 40.755668,-114.098984 40.755623,-114.09823 40.755601,-114.097528
40.755614,-114.095211 40.755606,-114.094307 40.755612,-114.090447 40.75562,-
114.083639 40.755589,-114.079599 40.75557,-114.067491 40.755541,-114.065279
40.755536,-114.064746 40.755525,-114.04451 40.75554,-114.043738 40.755541,-
114.043737 40.755502,-114.043736 40.755463,-114.04372 40.754919,-114.043772
40.74993,-114.043561 40.746899,-114.043486 40.74579,-114.043474 40.745643,-
114.043476 40.744303,-114.043479 40.741912,-114.043481 40.740851,-114.043481
40.740335,-114.043481 40.740265,-114.043481 40.740096,-114.043481 40.740041,-
114.043482 40.739687,-114.043485 40.738162,-114.043485 40.738008,-114.043501
40.736902,-114.043501 40.736686,-114.043503 40.735881,-114.043501 40.735806,-
114.043501 40.735461,-114.043501 40.732529,-114.043502 40.731996,-114.043502
40.731949,-114.043504 40.731519,-114.043504 40.731421,-114.043504 40.729113,-
114.043504 40.728462,-114.043595 40.727049,-114.043549 40.726198,-114.043543
40.726096,-114.043543 40.726033,-114.048071 40.725998,-114.0492 40.725989,-
114.052039 40.72597,-114.052461 40.725962,-114.054555 40.725945,-114.067632
40.725848,-114.071572 40.725819,-114.075327 40.725739,-114.076611 40.72574,-
114.079056 40.725743,-114.081055 40.725745,-114.092441 40.72565,-114.094021
40.725634,-114.112847 40.725489)))
```

The string is added to a set, and the *Feature* individual and the set are added to a hash map from individuals to geometry sets. The individual is marked as grounded.

*Step 6:* Ground the *UnnamedFeatures* with the *GroundedFeatureSets*. There are no *GroundedFeatureSets* yet, so this step fails.

*Step 7:* Ground the ungrounded *FeatureSets* with the *GroundedFeatures*. West Wendover is grounded, and is the only member of the first *FeatureSet*, so the *FeatureSet* can be grounded. A computational geometry function (JTS's GeometryFactory.createGeometryCollection()) is used to create a GeometryCollection with West Wendover's single geometry. The well-known text string for the GeometryCollection simply has "GEOMETRYCOLLECTION()" around the original geometry's well-known text string. The *FeatureSet* individual and a set containing the GeometryCollection are added to the hash map.

*Repeat Step 6:* Ground the *UnnamedFeatures* with the *GroundedFeatureSets*. The server determines that the ontology contains a triple that relates (NN city) to the newly grounded *FeatureSet*. (NN city) is an instance of the *ExistingFeature* class, so the server uses a PostGIS query. In this case, the

only property from the query ontology is *withinDistance*. The server is currently hard-coded to translate the *withinDistance* property to "ST_DWithin", and insert it into a query:

> "select asText(the_geom) from place where ST_DWithin(the_geom,GeomFromText('GEOMETRYCOLLECTION(MULTIPOLYGON(((-114.112847 40.725489,...*additional coordinates omitted*...)))',4326),.05) limit 25;"

The .05 is a distance threshold that is hard-coded for now, corresponding to .05 arc degrees in a spherical coordinate system. The "limit 25" ensures that no more than 25 results are returned. It is also hard-coded. The "4326" identifies the coordinate system, which the GIS uses to determine how to convert the WKT geometry string into its internal geometry representation. In the future, the hard-coded information will be expressed in a database schema ontology instead. This query returns two geometries, one for Wendover, UT, and one for West Wendover itself. The server removes the geometry for West Wendover because it has the same geometry as the reference geometry. The remaining geometry is added to a set, which is added to the map. The individual is marked as grounded.

*Repeat Step 7:* Ground the ungrounded *FeatureSet*s with the *GroundedFeature*s. The (NN city) individual is the only member of the *FeatureSet* (NP (DT the) (NN city)), so a computational geometry function is used to create a GeometryCollection with one geometry. The *FeatureSet* individual and the GeometryCollection are added to the map.

*Step 8:* Create an XML output string. All the *Feature*s are grounded, so the map from *Feature*s to geometry sets is translated into an XML string:

```
<ENTITY><NAME>West_Wendover </NAME>
<GML><gml:MultiPolygon srsName='0'>
  <gml:polygonMember>
    <gml:Polygon srsName='0'>
      <gml:outerBoundaryIs>
        <gml:LinearRing srsName='0'>
          <gml:coordinates>
-114.112847,40.725489 -114.112591,40.726539
… additional coordinates omitted …
          </gml:coordinates>
        </gml:LinearRing>
```

```
      </gml:outerBoundaryIs>
    </gml:Polygon>
  </gml:polygonMember>
</gml:MultiPolygon>
</GML></ENTITY>
<ENTITY><NAME>the city </NAME>
<GML><gml:MultiPolygon srsName='0'>
  <gml:polygonMember>
    <gml:Polygon srsName='0'>
      <gml:outerBoundaryIs>
        <gml:LinearRing srsName='0'>
          <gml:coordinates>
-114.043502,40.731996 -114.043501,40.732529
… additional coordinates omitted …
          </gml:coordinates>
        </gml:LinearRing>
      </gml:outerBoundaryIs>
    </gml:Polygon>
  </gml:polygonMember>
</gml:MultiPolygon>
</GML></ENTITY>
```

Each *FeatureSet* becomes a MultiPolygon in the XML string. The MultiPolygons for "West Wendover" and "the city" both consist of a single Polygon, with no holes.

*Step 9:* Display. The GeoCoder client converts the XML string to KML and opens it in Google Earth. Figure 5.3 shows the geometry of West Wendover in yellow and the geometry of the city near West Wendover in blue.


**Figure 5.3. The city near West Wendover in Google Earth**

| Step | Result |
|---|---|
| 1. Named Entity Tagging | the city near West_Wendover/LOCATION |
| 2. Parsing | (ROOT (NP (NP (DT the) (NN city)) (PP (IN near) (NP (NNP West_Wendover/LOCATION))))) |
| 3. Translation to Parse Ontology | (NN city) *rdf:type City*;<br>(NNP West_Wendover/LOCATION) *rdf:type NamedFeature* |
| 4. Application of Rules | (NP (NNP West_Wendover/LOCATION)) *rdf:type FeatureSet*;<br>(NP (NNP West_Wendover/LOCATION)) *hasFeature* (NNP West_Wendover/LOCATION);<br>(NP (DT the) (NN city)) *rdf:type FeatureSet*;<br>(NP (DT the) (NN city)) *hasFeature* (NN city);<br>(NN city) *near* (NP (NNP West_Wendover/LOCATION));<br>(NN city) *withinDistance* (NP (NNP West_Wendover/LOCATION));<br>(NN city) *rdf:type ExistingFeature* |
| 5. Ground the *NamedFeatures* | (NNP West_Wendover/LOCATION): 1 MultiPolygon |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | None |

| | |
|---|---|
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (NNP West_Wendover/LOCATION)): 1 GeometryCollection |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | (NN city): 1 MultiPolygon |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (DT the) (NN city)): 1 GeometryCollection |
| 8. Create an XML output string | <ENTITY><NAME>West_Wendover </NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML ></ENTITY><ENTITY><NAME>the city</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></ GML></ENTITY> |

**Table 5.1. GeoCoder's processing of "the city near West Wendover"**

## 5.2. Ambiguous Preposition: "the city around Harwood Heights"

Harwood Heights is an Illinois city that is completely surrounded by Chicago. Table 5.2 summarizes GeoCoder's processing of "the city around Harwood Heights". The name "Harwood Heights" is unambiguous, but the preposition "around" has multiple meanings. The relation ontology includes three senses of "around", meaning surrounding, near, and throughout, based on Merriam-Webster's online dictionary [24]. This example is similar to the first example, but demonstrates GeoCoder's ability to incorporate heuristics for disambiguating prepositional phrases. The steps that differ from the first example are described.

*Step 4:* Application of Rules. The server applies Jena rules as in the first example, but the rules file also includes a preposition disambiguation heuristic. The heuristic says that when an *around* property has a *City* as its first object, it should be interpreted as the *around-surrounding* sense. It is easy to add or modify heuristics. If the first object was a *Person*, it might make sense to prefer the *around-throughout* interpretation. This aspect of GeoCoder will become even more important when support for verbs is added. The heuristic rule adds the assertion

(NN city) *around-surrounding* (NP (NNP Harwood_Heights))

and the Pellet reasoner automatically infers the *properPartInverse* and *contains* properties.

*Repeat Step 6:* Ground the *UnnamedFeature*s with the *GroundedFeatureSet*s. The (NN city) individual is an instance of the *ExistingFeature* class, so the server uses a PostGIS query. The only property from the query ontology is *contains*. The server creates the query

> "select asText(the_geom) from place where contains(convexHull(the_geom),GeomFromText('MULTIPOLYGON(((-87.818961 41.96343,...*additional coordinates omitted...*)))',4326)) limit 25;"

The query returns the geometry of Chicago as the only result.

*Step 9:* Display. The GeoCoder client converts the XML string to KML and opens it in Google Earth. Figure 5.4 shows the geometry of Harwood Heights in yellow and the geometry of the city around Harwood Heights in blue.


**Figure 5.4. The city around Harwood Heights in Google Earth**

| Step | Result |
|---|---|
| 1. Named Entity Tagging | the city near Harwood_Heights/LOCATION |
| 2. Parsing | (ROOT (NP (NP (DT the) (NN city)) (PP (IN near) (NP (NNP Harwood_Heights/LOCATION))))) |
| 3. Translation to Parse Ontology | (NN city) *rdf:type City*;<br>(NNP Harwood_Heights/LOCATION) *rdf:type NamedFeature* |
| 4. Application of Rules | (NP (NNP Harwood_Heights/LOCATION)) *rdf:type FeatureSet*;<br>(NP (NNP Harwood_Heights/LOCATION)) *hasFeature* (NNP Harwood_Heights/LOCATION);<br>(NP (DT the) (NN city)) *rdf:type FeatureSet*;<br>(NP (DT the) (NN city)) *hasFeature* (NN city);<br>(NN city) *around* (NP (NNP Harwood_Heights/LOCATION));<br>(NN city) *around-surrounding* (NP (NNP Harwood_Heights/LOCATION));<br>(NN city) *properPartInverse* (NP (NNP Harwood_Heights/LOCATION));<br>(NN city) *contains* (NP (NNP Harwood_Heights/LOCATION));<br>(NN city) *rdf:type ExistingFeature* |

| | |
|---|---|
| 5. Ground the *NamedFeatures* | (NNP Harwood_Heights/LOCATION): 1 MultiPolygon |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | None |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (NNP Harwood_Heights/LOCATION)): 1 GeometryCollection |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | (NN city): 1 MultiPolygon |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (DT the) (NN city)): 1 GeometryCollection |
| 8. Create an XML output string | <ENTITY><NAME>Harwood_Heights </NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML ></ENTITY><ENTITY><NAME>the city</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></ GML></ENTITY> |

**Table 5.2. GeoCoder's processing of "the city around Harwood Heights"**

## 5.3. Ambiguous Phrase with Derived Feature: "the south of Boston"

The next example introduces two types of ambiguity and a *DerivedFeature*. First, the feature name "Boston" is ambiguous. The GeoCoder database currently contains three Bostons, in Massachusetts, Georgia, and Indiana. Second, the meaning of "south of" is ambiguous. It could refer to the southern part or to the area below the southern boundary. In this case, the correct interpretation is the southern part, which is a *DerivedFeature*. This example demonstrates GeoCoder's ability to derive a new feature geometry by performing a geometric operation on the geometry of a reference feature, and to order results for a *NamedFeature* based on a disambiguator in the ontology. Table 5.3 summarizes GeoCoder's processing of "the south of Boston".

*Step 4:* Application of Rules. The Pellet reasoner automatically infers that (NN south) is a *DerivedFeature*, and an instance of the *SouthTriangle* class from the query ontology. The sense of the preposition "of" hasn't been specified, so a heuristic rule could be applied to select the appropriate

sense. In this case, however, the preposition was already disambiguated when the phrase was parsed. Once "south" has been identified as a noun, there is only one possible interpretation of "of".

*Step 5:* Ground the *NamedFeature*. The server creates the SQL query

"select asText(the_geom) from place where name like 'Boston' order by area(the_geom) desc;"

The "order by" clause is present because an *orderBy* property has been added to the *NamedFeature* class in the domain ontology, with a value of area. The query returns three WKT strings, sorted from largest area to smallest area.

*Step 7:* Ground the ungrounded *FeatureSet*s with the *GroundedFeature*s. Boston is the only member of the first *FeatureSet*, so the *FeatureSet* can be grounded. A computation geometry function is used to create three GeometryCollections, one for each possible Boston geometry. The order is preserved, so the first GeometryCollection is the one for Boston, MA, which has the largest area. The *FeatureSet* individual and a set containing the GeometryCollections are added to the map.

*Repeat Step 6:* Ground the *UnnamedFeature*s with the *GroundedFeatureSet*s. The individual (NN south) is an instance of the *DerivedFeature* class, so the server uses a computational geometry function. The *SouthTriangle* class maps to an operation that creates a new geometry based on the centroid of the reference geometry and two 45° lines (to the southwest and southeast) from the centroid to the boundary. This operation is performed on each of the three possible geometries for Boston. Again, order is preserved. The set of results is added to the map, and the individual is marked as grounded.

*Repeat Step 7:* Ground the ungrounded *FeatureSet*s with the *GroundedFeature*s. The (NN south) individual is the only member of (NP (DT the) (NN south)), so a computational geometry function is used to create three GeometryCollections with one geometry each.

*Step 9:* Display. The GeoCoder client converts the XML string to KML and opens it in Google Earth. Figure 5.5 shows the first result: the geometry of Boston, MA in yellow and the computed geometry for "the south" in blue. This isn't a very good interpretation of the "south" concept, but in the future GeoCoder will have additional functions for other interpretations, and will be able to select the best function based on the shape of the geometry or based on heuristics and semantic information.

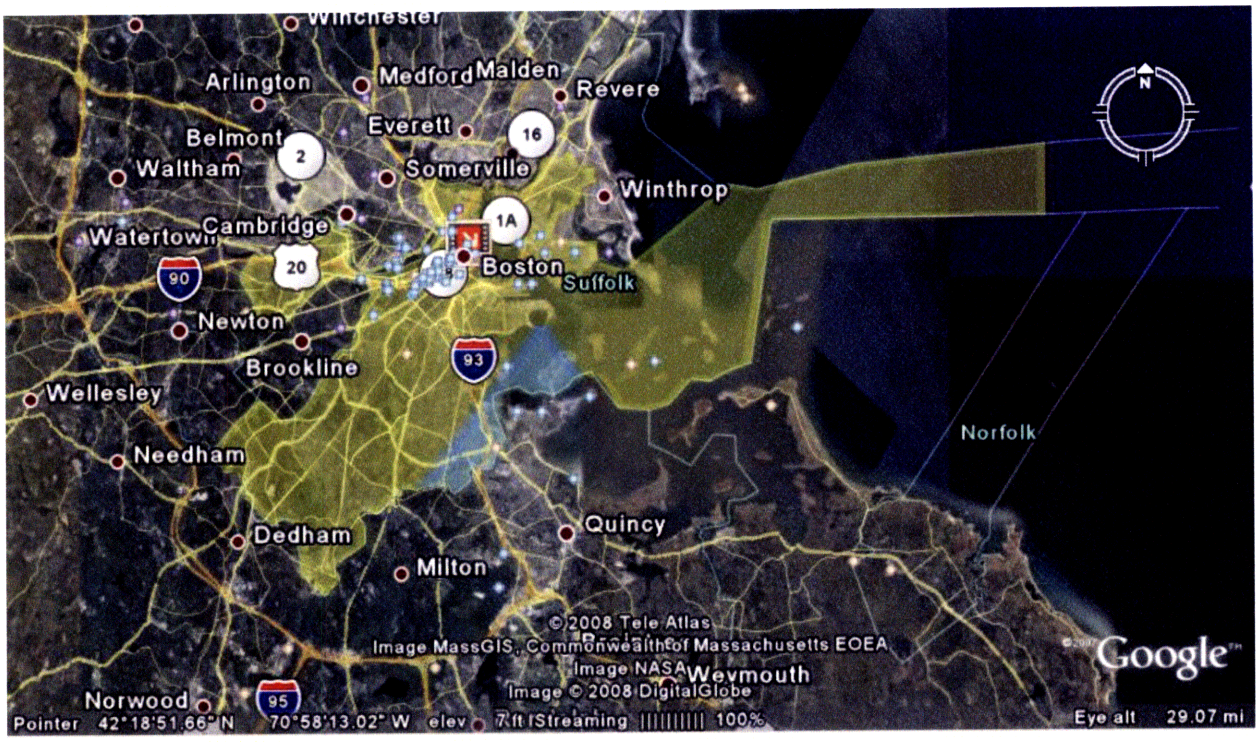


**Figure 5.5. First result for "the south of Boston" in Google Earth**

Figure 5.6 shows the third result, with the geometry of Boston, IN in yellow and the computed geometry for "the south" in blue. This Boston's geometry is square, so the interpretation of "south" is reasonable.

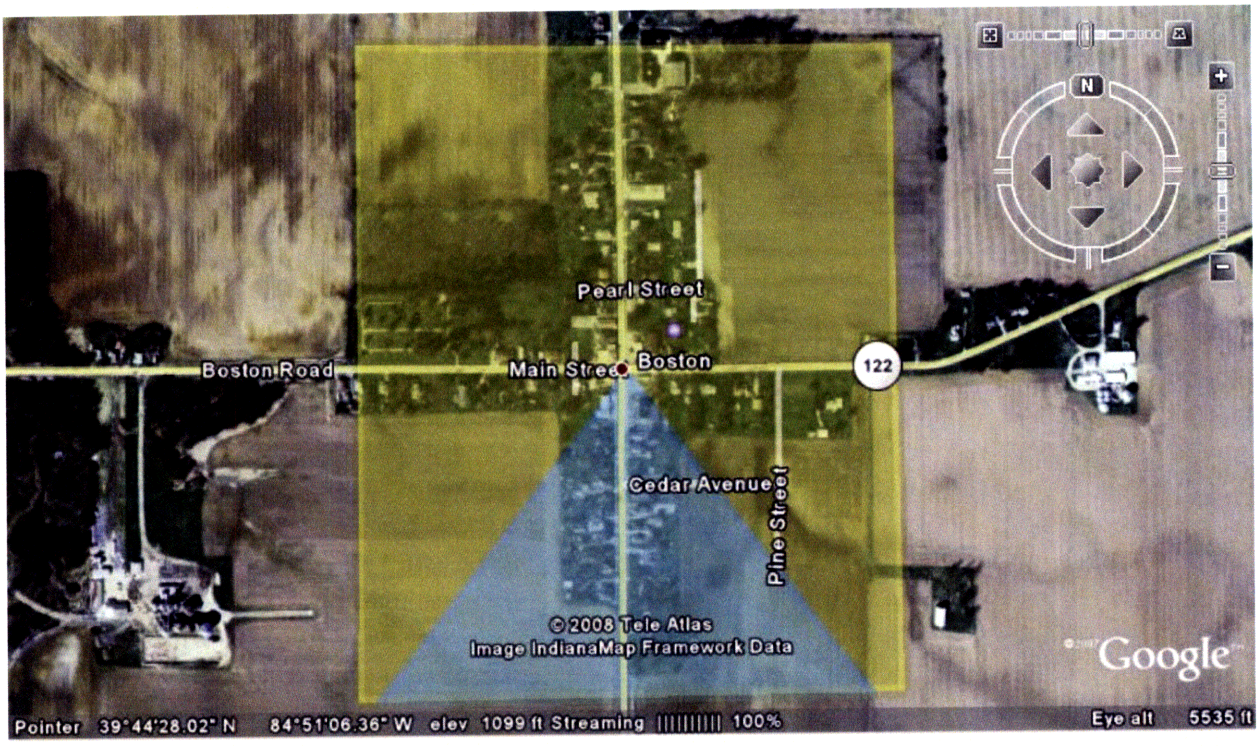


**Figure 5.6. Third result for "the south of Boston" in Google Earth**

| Step | Result |
|---|---|
| 1. Named Entity Tagging | the south of Boston/LOCATION |
| 2. Parsing | (ROOT (NP (NP (DT the) (NN south)) (PP (IN of) (NP (NNP Boston/LOCATION))))) |
| 3. Translation to Parse Ontology | (NN south) *rdf:type South*;<br>(NNP Boston/LOCATION) *rdf:type NamedFeature* |
| 4. Application of Rules | (NP (NNP Boston/LOCATION)) *rdf:type FeatureSet*;<br>(NP (NNP Boston/LOCATION)) *hasFeature* (NNP Boston/LOCATION);<br>(NP (DT the) (NN south)) *rdf:type FeatureSet*;<br>(NP (DT the) (NN south)) *hasFeature* (NN south);<br>(NN south) *of* (NP (NNP Boston/LOCATION));<br>(NN city) *rdf:type DerivedFeature*;<br>(NN city) *rdf:type SouthTriangle* |
| 5. Ground the *NamedFeatures* | (NNP Boston/LOCATION): 3 MultiPolygons |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | None |

| | |
|---|---|
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (NNP Boston/LOCATION)): 3 GeometryCollections |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | (NN south): 3 Polygons |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (DT the) (NN south)): 3 GeometryCollections |
| 8. Create an XML output string | <ENTITY><NAME>the south</NAME><GML><gml:Polygon>...</gml:Polygon></GML></ENTITY><ENTITY><NAME>the south</NAME><GML><gml:Polygon>...</gml:Polygon></GML></ENTITY><ENTITY><NAME>the south</NAME><GML><gml:Polygon>...</gml:Polygon></GML></ENTITY><ENTITY><NAME>Boston </NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML></ENTITY><ENTITY><NAME>Boston </NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML></ENTITY><ENTITY><NAME>Boston </NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML></ENTITY> |

**Table 5.3. GeoCoder's processing of "the south of Boston"**

## 5.4. Preposition Chain with Existing and Derived Features: "the city near the north of New York"

This example demonstrates GeoCoder's ability to process preposition chains as well as single prepositions. This requires the composition of geometric functions and PostGIS queries. While this example involves one *ExistingFeature* and one *DerivedFeature*, GeoCoder can process arbitrary combinations of *ExistingFeatures* and *DerivedFeatures*, provided the Stanford Parser is able to determine a correct parse tree. "New York" is unambiguous in our database, and refers to New York City. Disambiguating between the city and the state is an interesting problem, which could be dealt with through heuristics, or by returning both results and giving the user a choice. Table 5.4 summarizes GeoCoder's processing of "the city near the north of New York".

*Step 2:* Parsing. The tagged phrase is passed to the parser, which returns the parse tree in Figure 5.7.

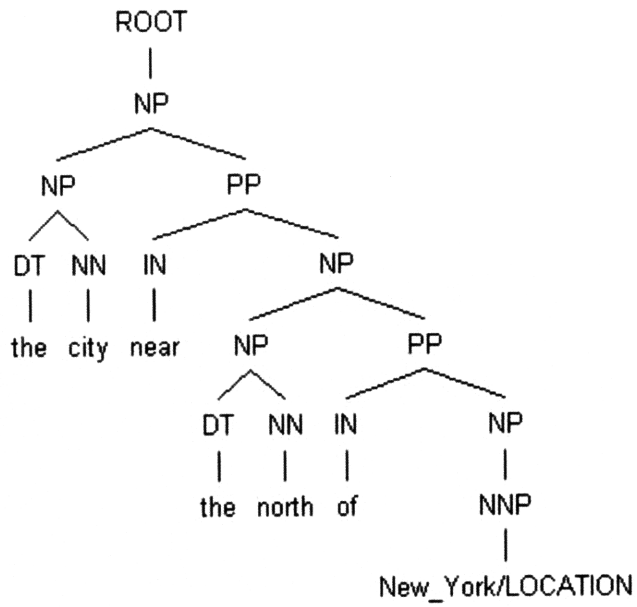


**Figure 5.7. Parse tree for "the city near the north of New_York/LOCATION"**

*Step 4:* Application of Rules. Two of the subtrees match the NP-PP rule, producing

(NN city) *near* (NP (DT the) (NN north))

and

(NN north) *of* (NP (NNP New_York/LOCATION))

*Repeat Step 6:* Ground the *UnnamedFeature*s with the *GroundedFeatureSet*s. The server tries to ground (NN city) with the computed geometry for (NP (DT the) (NN north)). The query is like the query in the first example, but now the results are sorted by area and the 25 largest are returned. Sorted by distance to New York is another heuristic that would be appropriate for *near* phrases, and will be implemented in a future version of GeoCoder. The largest result by area is the geometry for New York itself. The geometry of New York *is* near its own northern part, so GeoCoder does not remove the result.

It might, however, be appropriate to remove it, as people won't say "the city near the north of New York" when they mean New York itself.

*Step 9:* Display. The GeoCoder client converts the XML string to KML and opens it in Google Earth. Figure 5.8 shows New York in blue, "the north" in green, and 24 results for "the city near the north of New York" (excluding the New York result) in yellow.


**Figure 5.8. Results for "the city near the north of New York" in Google Earth**

| Step | Result |
|---|---|
| 1. Named Entity Tagging | the city near the north of New_York/LOCATION |
| 2. Parsing | (ROOT (NP (NP (DT the) (NN city)) (PP (IN near) (NP (NP (DT the) (NN north)) (PP (IN of) (NP (NNP New_York/LOCATION)))))) |
| 3. Translation to Parse Ontology | (NN city) *rdf:type City*;<br>(NN north) *rdf:type North*;<br>(NNP New_York/LOCATION) *rdf:type NamedFeature* |

| | |
|---|---|
| 4. Application of Rules | (NP (NNP New_York/LOCATION)) *rdf:type FeatureSet*;<br>(NP (NNP New_York/LOCATION)) *hasFeature* (NNP New_York/LOCATION);<br>(NP (DT the) (NN city)) *rdf:type FeatureSet*;<br>(NP (DT the) (NN city)) *hasFeature* (NN city);<br>(NP (DT the) (NN north)) *rdf:type FeatureSet*;<br>(NP (DT the) (NN north)) *hasFeature* (NN north);<br>(NN north) *of* (NP (NNP New_York/LOCATION));<br>(NN city) *near* (NP (DT the) (NN north));<br>(NN city) *withinDistance* (NP (DT the) (NN north));<br>(NN north) *rdf:type DerivedFeature*;<br>(NN city) *rdf:type ExistingFeature*;<br>(NN north) *rdf:type NorthTriangle* |
| 5. Ground the *NamedFeatures* | (NNP New_York/LOCATION): 1 MultiPolygon |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | None |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (NNP New_York/LOCATION)): 1 GeometryCollection |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | (NN north): 1 Polygon |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (DT the) (NN north)): 1 GeometryCollection |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | (NN city): 25 MultiPolygons |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (DT the) (NN city)): 25 GeometryCollections |
| 8. Create an XML output string | <ENTITY><NAME>the north</NAME><GML><gml:Polygon>...</gml:Polygon></GML></ENTITY><ENTITY><NAME>New York</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML></ENTITY><ENTITY><NAME>the city</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML></ENTITY><ENTITY><NAME>the city</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML></ENTITY><ENTITY><NAME>the city</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML></ENTITY>... |

**Table 5.4. GeoCoder's processing of "the city near the north of New York"**

## 5.5. PostGIS Lookup with Feature Set: "the city between Seattle and Tacoma"

This example introduces a ternary between relation. GeoCoder creates a *FeatureSet* for "Seattle and Tacoma", and treats the relation as a binary relation between a *Feature* and a *FeatureSet*. Both city names are unambiguous. This example also demonstrates a more complicated PostGIS function that has been implemented in GeoCoder. Table 5.5 summarizes GeoCoder's processing of "the city between Seattle and Tacoma".

*Step 1:* Named Entity Tagging. The tagger recognizes both place names and gives them separate tags: "the city between Seattle/LOCATION and Tacoma/LOCATION".

*Step 2:* Parsing. The parser produces the tree in Figure 5.9.

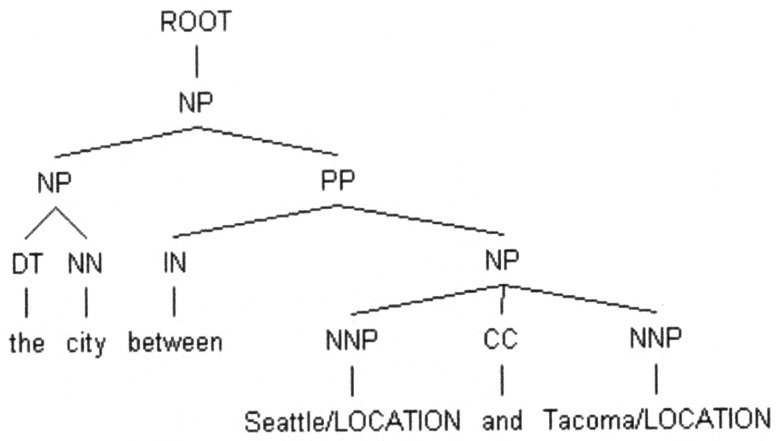


**Figure 5.9. Parse tree for "the city between Seattle/LOCATION and Tacoma/LOCATION"**

*Step 4:* Application of Rules. The (NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION)) individual becomes a single *FeatureSet*, with *hasFeature* (NNP Seattle/LOCATION) and *hasFeature* (NNP Tacoma/LOCATION). The NP-PP rule produces

(NN city) *between* (NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION))

and the *intersects-buffer-convexHull* property is inferred from *between*.

*Step 5:* Ground the *NamedFeatures*. The server iterates through the two *NamedFeatures*. It creates a query and gets one MultiPolygon for each.

*Step 7:* Ground the ungrounded *FeatureSets* with the *GroundedFeatures*. The *FeatureSet* (NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION)) has two member *Features*, both of which are grounded. JTS is used to create a GeometryCollection containing both of their geometries. Both geometries are unambiguous in this case, but if either *Feature* had more than one possible geometry, the server would create one GeometryCollection for every possible combination of one geometry from each *Feature*. For example, if Seattle had 5 possible geometries and Tacoma had 3, there would be 15 GeometryCollections.

*Step 6:* Ground the *UnnamedFeatures* with the *GroundedFeatureSets*. The server constructs an *intersects-buffer-convexHull* query:

> "select asText(the_geom) from place where intersects(the_geom, buffer(convexHull(GeomFromText('GEOMETRYCOLLECTION(...*coordinates omitted*...)',4326)), 0.1*ST_Distance(GeometryN(GeomFromText('GEOMETRYCOLLECTION(...*coordinates omitted*...)',4326),1),GeometryN(GeomFromText('GEOMETRYCOLLECTION(...*coordinates omitted*...)',4326),2))))order by area(the_geom) desc limit 25;"

First, this query computes the convex hull of the combined geometries of Seattle and Tacoma, as shown in Figure 5.10.


**Figure 5.10. Convex hull of Seattle and Tacoma**

Next, the query uses PostGIS's buffer function to increase the size of the convex hull based on the distance between Seattle and Tacoma. The GeometryN function is used to get the first and second geometries from the GeometryCollection, and the ST_Distance function is used to compute the distance between them. The buffer function then expands the geometry of the convex hull to include everything within a certain distance of the convex hull. In this case, the distance is 0.1 times the distance between Seattle and Tacoma. The expanded geometry is shown in Figure 5.11.

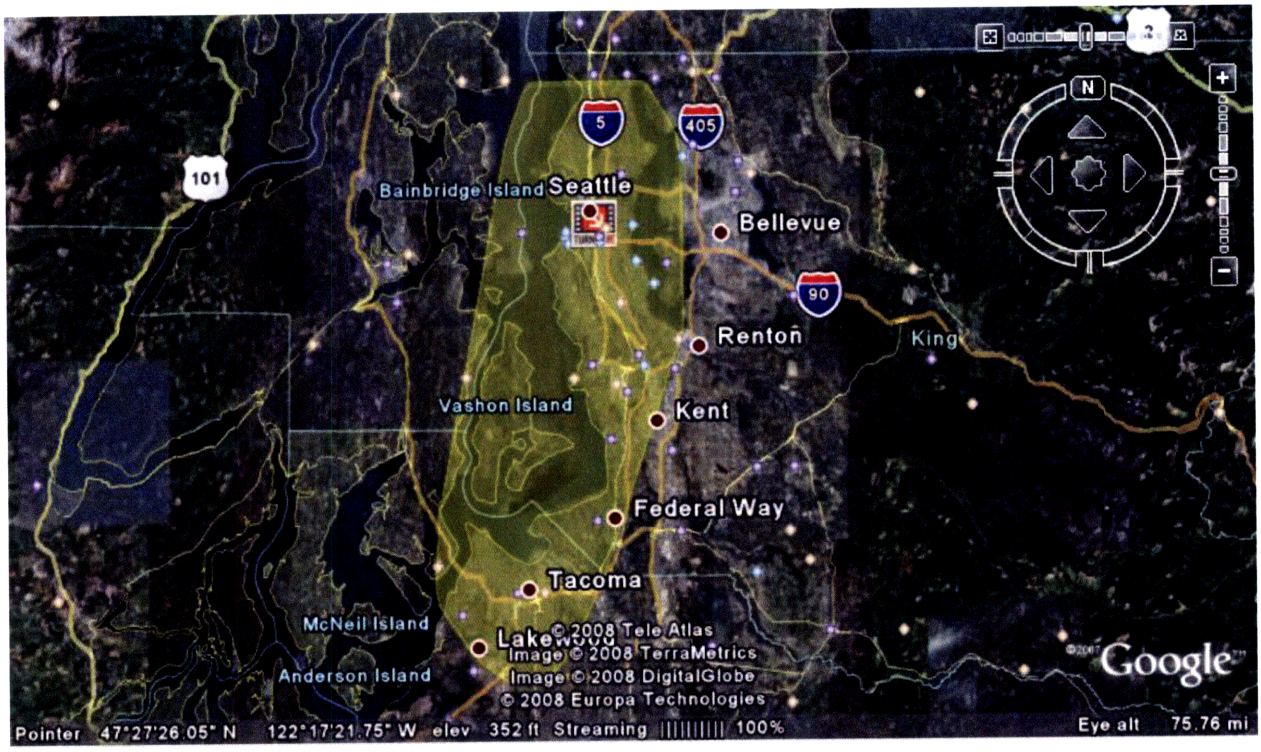


**Figure 5.11 Expanded hull of Seattle and Tacoma**

The convex hull is extended equally far in all directions, so that it overlaps with some cities north of Seattle or south of Tacoma. A better (but much more complicated) query would expand the hull most halfway between the cities, and not at all at the ends. Finally, the intersects function finds the geometries that intersect the expanded hull.

*Step 8:* Create an XML output string. The <GML> element for Seattle and Tacoma includes a MultiPolygon with two MultiPolygons as its members.

*Step 9:* Display. The GeoCoder client converts the XML string to KML and opens it in Google Earth. Figure 5.12 shows Seattle and Tacoma in blue, and the 23 results for "the city between Seattle and Tacoma" in yellow.

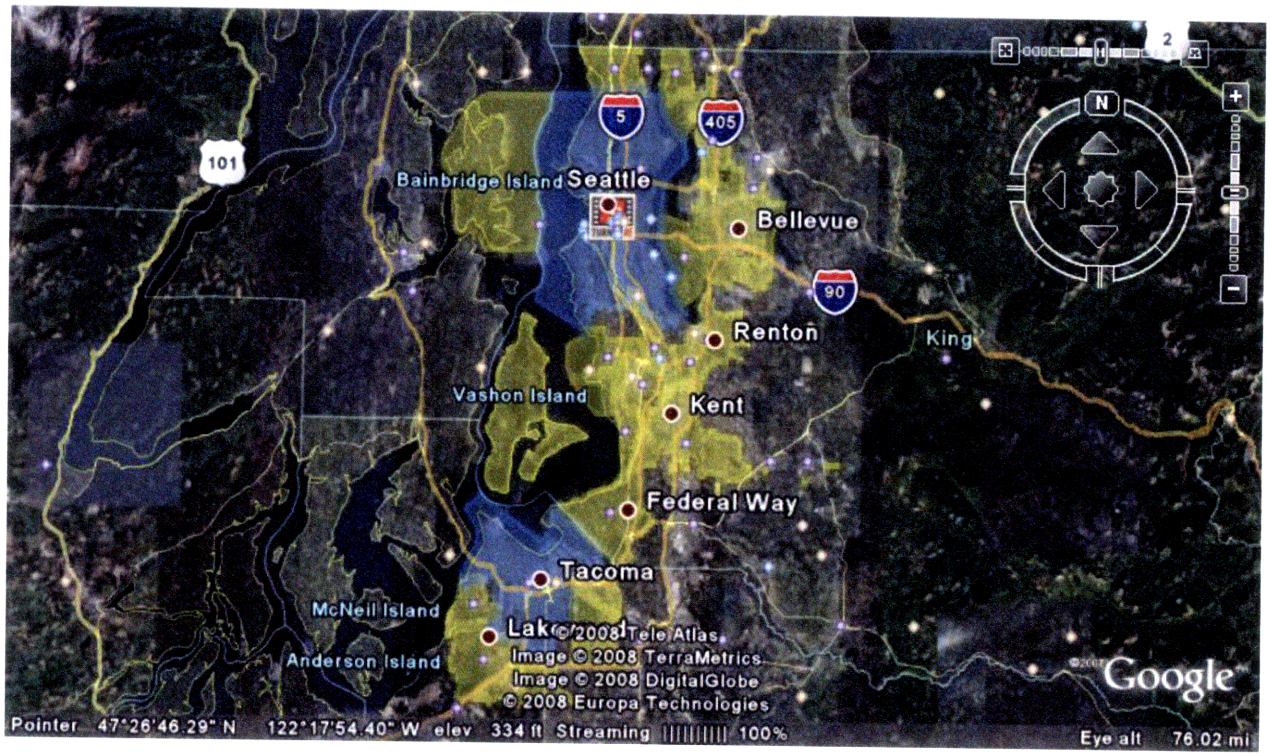


**Figure 5.12. Results for "the city between Seattle and Tacoma" in Google Earth**

| Step | Result |
|---|---|
| 1. Named Entity Tagging | the city between Seattle/LOCATION and Tacoma/LOCATION |
| 2. Parsing | (ROOT (NP (NP (DT the) (NN city)) (PP (IN between) (NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION))))) |
| 3. Translation to Parse Ontology | (NN city) *rdf:type City*;<br>(NN Seattle/LOCATION) *rdf:type NamedFeature*;<br>(NNP Tacoma/LOCATION) *rdf:type NamedFeature* |
| 4. Application of Rules | (NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION)) *rdf:type FeatureSet*;<br>(NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION)) *hasFeature* (NNP Seattle/LOCATION);<br>(NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION)) *hasFeature* (NNP Tacoma/LOCATION);<br>(NP (DT the) (NN city)) *rdf:type FeatureSet*;<br>(NP (DT the) (NN city)) *hasFeature* (NN city);<br>(NN city) *between* (NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION));<br>(NN city) *intersect-buffer-convexHull* (NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION));<br>(NN city) *rdf:type ExistingFeature*; |

| | |
|---|---|
| 5. Ground the *NamedFeatures* | (NNP Seattle/LOCATION): 1 MultiPolygon<br>(NNP Tacoma/LOCATION): 1 MultiPolygon |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | None |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (NNP Seattle/LOCATION) (CC and) (NNP Tacoma/LOCATION)): 1 GeometryCollection |
| 6. Ground the *UnnamedFeatures* with the *GroundedFeatureSets* | (NN city): 23 MultiPolygons |
| 7. Ground the ungrounded *FeatureSets* with the *GroundedFeatures* | (NP (DT the) (NN city)): 23 GeometryCollections |
| 8. Create an XML output string | <ENTITY><NAME>Seattle and Tacoma </NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></GML ></ENTITY><ENTITY><NAME>the city</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></ GML></ENTITY><ENTITY><NAME>the city</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></ GML></ENTITY><ENTITY><NAME>the city</NAME><GML><gml:MultiPolygon>...</gml:MultiPolygon></ GML></ENTITY>... |

**Table 5.5. GeoCoder's processing of "the city between Seattle and Tacoma"**

# Chapter 6

# Future Work

## 6.1. Paths

Jackendoff [19] distinguishes places and paths as the basic senses of a spatial prepositional phrase. Places refer to regions or points, and paths are bounded paths, directions, or routes. GeoCoder's ontologies only support places, but adding a representation of paths would enable GeoCoder to process many more phrases, such as "from Boston to New York" or "along the river". The path representation would introduce additional ambiguity. For example, "between Boston and New York" could refer to a path ("traveled between Boston and New York"), a place ("the city between Boston and New York"), or a region ("the area between Boston and New York").

The first step will be to introduce paths to the domain ontology as *DerivedFeature*s or *ExistingFeature*s. If the database contains natural paths, such as roads, train routes, or hiking trails,

subclasses of *ExistingFeature* could be created to represent them in the ontology. Words like "path" and "route" could become *DerivedFeature*s. Computational geometry functions could be used to compute path geometries if there is no natural path in the database. Different domain ontologies could select different computational geometry functions. For example, an air travel domain should allow a path that goes directly over water, but a land travel ontology could find a path that goes around instead.

## 6.2. Verbs

A complete GeoCoder system should be able to understand spatial relationships that are expressed through verbs as well as spatial prepositions. Phrases with verbs are more varied in both structure and meaning than those with spatial prepositions. Processing verbs will require additional rules for transforming NP-VP structures into relationships between features. Verbs can describe spatial relationships directly ("Chicago surrounds Harwood Heights") or by using spatial prepositions ("Boston is in Massachusetts"). Verbs (especially motion verbs) can be used to indicate paths ("We drove from Boston to New York"), and will provide additional opportunities for disambiguation. For example, "is" could indicate the place sense of "between" and "went" could indicate the path sense. Sometimes, however, motion verbs cause prepositions to be ambiguous between place and path senses. Jackendoff [19] gives the example "The mouse went under the table", in which "under" could indicate that the mouse went *to* the place under the table, or that the mouse went *via* the path under the table. Finally, verbs can help with disambiguation by providing additional information about the domain. For example, the verb "swim" might indicate that some of the features in the sentence are likely to be bodies of water.

## 6.3. Integration with WordNet

WordNet [30] is an English lexicon project that organizes words into a semantic hierarchy. It can be used to look up synonyms, hypernyms (superclasses), and hyponyms (subclasses) of a word. For example, "pond" is a hyponym of "lake", which is a hyponym of "body of water". This information could be used to match words to classes in the domain ontology. If a phrase contains the word "pond" and the domain ontology doesn't have a *Pond* class, GeoCoder could look up hyponyms and determine that "pond" can be an instance of the *Lake* class instead. This would make it easier to develop new domain ontologies, because only high-level classes would need to be included. Additionally, WordNet information could be used to select a domain ontology to use. If there are domain ontologies for water travel and air travel, for example, GeoCoder could search WordNet for a word like "flew" and "plane" and determine that the air travel domain is more appropriate. This way, different domain ontologies can include different information and heuristics for the same feature types.

## 6.4. Database Schema Ontology

The database schema ontology will add a level of abstraction between GeoCoder and the GIS database, enabling GeoCoder to work with arbitrary databases or even with multiple databases or database tables simultaneously. In the current version, the mapping between the query ontology and the actual database attributes and functions is hard-coded into the Java functions that access the database. The database schema ontology will be used to describe the capabilities of the database, and the exact syntax of database functions. For example, representations of the buffer and convexHull PostGIS functions will include information about their arguments. The names of attributes like "the_geom" and of tables like "place" will also be included.

## 6.5. Parse Ambiguity

Currently, the GeoCoder server stores a set of possible GeometryCollections for each *FeatureSet*, and can iteratively remove the ones that are found to be impossible. It would be possible to store a similar set of possible parse trees for a phrase, so that GeoCoder would not have to rely on the accuracy of the parser. For example, the phrase "the park near the lake in Boston" has two possible parse trees, shown in Figure 6.1.

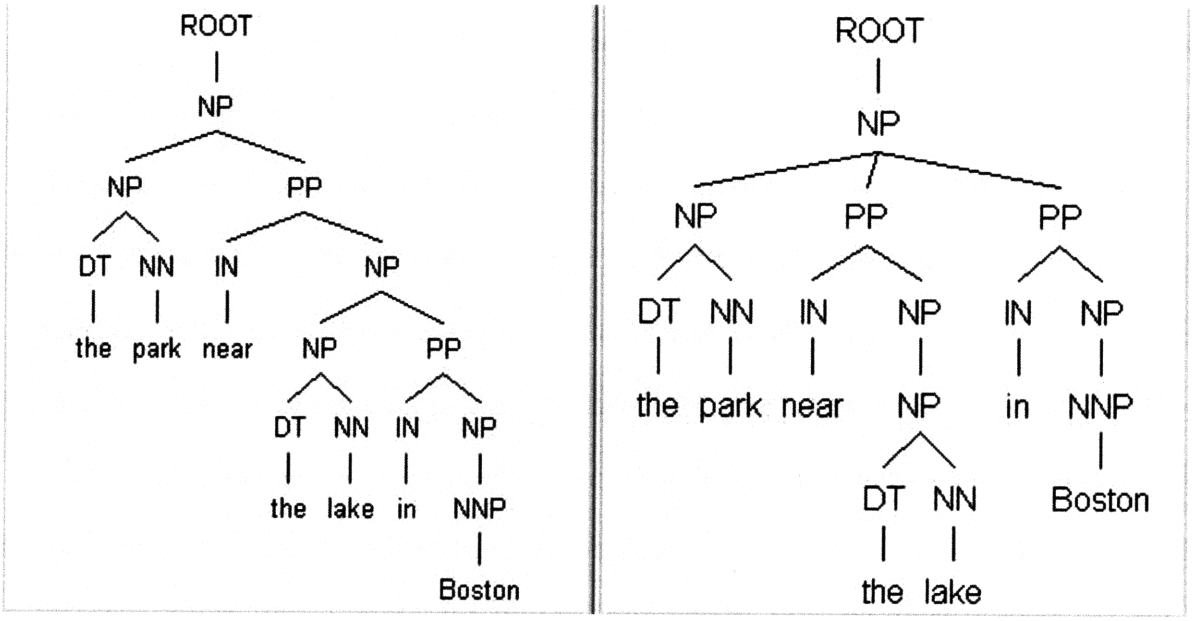


**Figure 6.1. Possible parse trees for "the park near the lake in Boston"**

The tree on the left indicates that the lake is in Boston, but the park might not be, while the tree on the right indicates that the park is in Boston, but the lake might not be. If both possible parses were stored in a set, information from the GIS could be used to disambiguate them at later processing stages.

## 6.6. Additional Prepositions, Existing Features, and Derived Features

The simplest way to improve GeoCoder is to expand the set of words and concepts that can be handled. Currently, GeoCoder only supports four prepositions. There are about 80 spatial prepositions in English, and many of them could be added to GeoCoder without any additional improvements. Some will require a path representation or three-dimensional space, and those could be added to later versions as well. Adding prepositions will also require adding additional PostGIS functions to the query ontology. The set of existing features needs to be expanded in both the actual database and the domain ontology. The database used in the examples only includes cities, but features like bodies of water and parks could make GeoCoder more interesting and useful. The only derived features in the current version are the four cardinal directions, but additional feature types like centers and boundaries could be added to the domain ontology and to the computational geometry function library. Also, new geometric functions could be developed to support more complicated partitioning of a region, and even partitioning based on the region's shape. Adding more PostGIS queries and computational geometry functions will enable the creation of better heuristics for complicated concepts like "between". Also, the distance thresholds for *withinDistance* queries are currently hard-coded in the Java function that accesses the database, so "near" is interpreted based on a fixed distance threshold. It would be better to use a data property in the query ontology to specify the threshold, or to develop a more complex metric based on e.g. feature area. Similarly, the limit on the number of results to return is hard-coded, and it would be better to set it based on the user's preferences. In addition, GIS queries could be extended to include feature types. If different feature types are in different tables in the database, the queries could allow for different table names. The *NamedFeature* class could be extended to include feature types, such as *City* or *River*, which could be used to improve the name-based queries to the GIS.

The queries could also be used to find entries that are only partial matches to the names (e.g. "Boston" would also match "Boston Harbor"), and sort based on similarity or other metrics.

# Chapter 7

# Conclusion

Geocoder is a spatial reasoning system that grounds and disambiguates locations mentioned in an input phrase. GeoCoder distinguishes itself from other grounding and disambiguation systems in several ways:

- GeoCoder grounds and disambiguates at the phrase level rather than the word level, and uses spatial relationships mentioned in the text rather than simple textual proximity to determine whether two location references are related.
- GeoCoder interprets the parses with rules that match mid-level sentence structure patterns, allowing the use of a statistical parser rather than a deterministic grammar to process a wide range of sentence structures.
- GeoCoder uses a set of modular, extensible ontologies, so that lower-level ontologies can be extended or modified (by a human, or even by machine learning) without requiring changes to the others.

- GeoCoder implements disambiguation heuristics in the ontologies, rather than in Java code, so that they are visible and editable by humans and logical reasoners.
- GeoCoder avoids the scalability problems involved in representing actual geospatial entities in the ontology, by first computing relationships between entities and then using a combination of database queries and computational geometry functions to ground the location references based on their relationships.
- GeoCoder takes an iterative approach to disambiguation, in which a set of possible geometries for a location reference are stored, and geometries are removed at later iterations if they are found to be inconsistent with the phrase and the contents of the GIS.

# Appendix A.1

# Parse Tree Traversal Algorithm

```
AddToOntology(parsetree)
      AddIndividual(parsetree)
      AddTriple(parsetree,"rdf:type",Label(parsetree))
      For each childtree in parsetree
            AddIndividual(childtree)
            AddTriple(childtree,"rdf:type",Label(childtree))
            AddTriple(parsetree,"child",childtree)
            If IsPreterminal(childtree)
                  If HasLocationTag(childtree)
                        AddTriple(childtree,"rdf:type","NamedFeature")
                  If IsNoun(childtree) or IsPreposition(childtree)
                        AddTriple(childtree,"rdf:type",Terminal(childtree))
            Else
                  AddToOntology(childtree)
```

# Appendix A.2

# Grounding Algorithm

```
GroundFeatures()
      For each namedfeature
            GroundWithGIS(namedfeature)
            MarkGrounded(namedfeature)
      Do    For each ungroundedfeature
                  FindFirstMatchingTriple(ungroundedfeature, relation,
                  groundedfeature)
                  GroundWithReferenceFeature(ungroundedfeature,
                  groundedfeature)
                  MarkGrounded(ungroundedfeature)
            For each ungroundedset
                  If AllMembersGrounded(ungroundedset)
                        GroundWithJTS(ungroundedset)
                        MarkGrounded(ungroundedset)
      While SomethingGrounded
```

# Appendix A.3

# Sample XML output string

The following is the server's output for "the south of Boston". There are three geometries for "the south" and three geometries for "Boston".

```
<ENTITY><NAME>the south </NAME><GML><gml:Polygon srsName='0'>
  <gml:outerBoundaryIs>
    <gml:LinearRing srsName='0'>
      <gml:coordinates>
-70.9966409938302,42.31231347574462 -70.998593,42.311998
        -71.002093,42.308698 -71.003541,42.30804
        … 132 lines of coordinates omitted …
        -70.9966409938302,42.31231347574462
      </gml:coordinates>
    </gml:LinearRing>
  </gml:outerBoundaryIs>
</gml:Polygon>
</GML></ENTITY>
<ENTITY><NAME>the south </NAME><GML><gml:Polygon srsName='0'>
  <gml:outerBoundaryIs>
    <gml:LinearRing srsName='0'>
```

```
      <gml:coordinates>
-83.796287,30.781049 -83.799579,30.781092
        -83.799658,30.781092 -83.80092932667594,30.781054127916175
        -83.78997223834466,30.79201121624746 -
83.77920263362738,30.781241611530174
        -83.785507,30.781181 -83.789939,30.781121
        -83.796287,30.781049
      </gml:coordinates>
    </gml:LinearRing>
  </gml:outerBoundaryIs>
</gml:Polygon>
</GML></ENTITY>
<ENTITY><NAME>the south </NAME><GML><gml:Polygon srsName='0'>
  <gml:outerBoundaryIs>
    <gml:LinearRing srsName='0'>
      <gml:coordinates>
-84.84823167440614,39.737528396833135 -84.85189,39.737492
        -84.85545632599117,39.73745787689121 -
84.8518087402277,39.74110546265469
        -84.84823167440614,39.737528396833135
      </gml:coordinates>
    </gml:LinearRing>
  </gml:outerBoundaryIs>
</gml:Polygon>
</GML></ENTITY>
<ENTITY><NAME>Boston </NAME><GML><gml:MultiPolygon srsName='0'>
  <gml:polygonMember>
    <gml:Polygon srsName='0'>
      <gml:outerBoundaryIs>
        <gml:LinearRing srsName='0'>
          <gml:coordinates>
-71.189597,42.281167 -71.18974,42.281236
            -71.190044,42.281415 -71.190185,42.281515
            … 1500 lines of coordinates omitted …
            -71.189597,42.281167
          </gml:coordinates>
        </gml:LinearRing>
      </gml:outerBoundaryIs>
    </gml:Polygon>
  </gml:polygonMember>
</gml:MultiPolygon>
</GML></ENTITY>
<ENTITY><NAME>Boston </NAME><GML><gml:MultiPolygon srsName='0'>
  <gml:polygonMember>
    <gml:Polygon srsName='0'>
      <gml:outerBoundaryIs>
        <gml:LinearRing srsName='0'>
          <gml:coordinates>
-83.796287,30.781049 -83.799579,30.781092
            -83.799658,30.781092 -83.801605,30.781034
            -83.802444,30.781036 -83.80265,30.781036
            -83.802628,30.783464 -83.802582,30.787666
            -83.802513,30.793365 -83.802418,30.794597
            -83.802393,30.794873 -83.802406,30.795017
            -83.802452,30.7988 -83.802391,30.802975
            -83.791618,30.802916 -83.784744,30.803051
            -83.777473,30.802999 -83.777451,30.800013
```

```
            -83.777623,30.793499 -83.777641,30.792803
            -83.777646,30.792545 -83.777565,30.792227
            -83.777496,30.792017 -83.777328,30.790783
            -83.777214,30.786886 -83.77729,30.78126
            -83.785507,30.781181 -83.789939,30.781121
            -83.796287,30.781049
          </gml:coordinates>
        </gml:LinearRing>
      </gml:outerBoundaryIs>
    </gml:Polygon>
  </gml:polygonMember>
</gml:MultiPolygon>
</GML></ENTITY>
<ENTITY><NAME>Boston </NAME><GML><gml:MultiPolygon srsName='0'>
  <gml:polygonMember>
    <gml:Polygon srsName='0'>
      <gml:outerBoundaryIs>
        <gml:LinearRing srsName='0'>
          <gml:coordinates>
-84.855758,39.742021 -84.855762,39.744778
            -84.851936,39.744724 -84.8479,39.744724
            -84.847847,39.741137 -84.847787,39.738751
            -84.847769,39.737533 -84.85189,39.737492
            -84.855757,39.737455 -84.855748,39.741161
            -84.855758,39.742021
          </gml:coordinates>
        </gml:LinearRing>
      </gml:outerBoundaryIs>
    </gml:Polygon>
  </gml:polygonMember>
</gml:MultiPolygon>
</GML></ENTITY>
```

# References

[1] "About GeoNames," URL: http://www.geonames.org/about.html [accessed November 30, 2007].

[2] "PostGIS: Home," URL: http://www.postgis.org [accessed May 21, 2008].

[3] Amitay, E., Har'El, N., Sivan, R., and Soffer, A., "Web-a-where: Geotagging Web Content," *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Association for Computing Machinery, New York, July 2004, pp. 273-280.

[4] Axelrod, A. E., "On Building a High Performance Gazetteer Database," *Proceedings of the HLT-NAACL 2003 Workshop on Analysis of Geographic References*, Vol. 1, Association for Computational Linguistics, Morristown, NJ, 2003, pp. 63-68.

[5] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F. (eds.), *The Description Logic Handbook: Theory, Implementation, and Applications*, Cambridge University Press, Cambridge, 2003.

[6] Bikel, D. M., Miller, S., Schwartz, R., and Weischedel, R., "Nymble: A High-Performance Learning Name-Finder," *Proceedings of the Fifth Conference on Applied Natural Language Processing*, Morgan Kaufman Publishers, San Francisco, April 1997, pp. 194-201.

[7] Cohn, A. G. and Gotts, N. M., "The Egg-Yolk Representation of Regions with Indeterminate Boundaries," *Proceedings, GISDATA Specialist Meeting on Geographical Objects with Undetermined Boundaries*, Francis Taylor, 1996, pp. 171-187.

[8] Dickinson, I., "The Jena Ontology API," URL: http://jena.sourceforge.net/ontology/index.html [accessed May 21, 2008].

[9] Dolbear, C., Hart, G., and Goodwin, J., "From Theory to Query: Using Ontologies to Make Explicit Imprecise Spatial Relationships for Database Querying," *Proceedings of the 2007 Conference on Spatial Information Theory*, Melbourne, September 2007.

[10] Egenhofer, M. J. and Herring, J. R., "Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases," National Center for Geographic Information and Analysis, University of California, Santa Barbara, Technical Report 90-12, 1990.

[11] El-Geresy, B. A. and Abdelmoty, A. I., "Towards a General Theory for Modeling Qualitative Space," *International Journal on Artificial Intelligence Tools*, Vol. 11, No. 3, pp. 347-367.

[12] Finkel, J. R., Grenager, T., and Manning, C., "Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling," *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Morristown, NJ, 2005, pp. 363-370.

[13] Fu, G., Jones, C. B., and Abdelmoty, A. I., "Ontology-Based Spatial Query Expansion in Information Retrieval," *On the Move to Meaningful Internet Systems: ODBASE*, 2005, pp. 1466-1482.

[14] Galton, A. and Hood, J., "Anchoring: A New Approach to Handling Indeterminate Location in GIS," *spatial Information Theory: Proceedings of International Conference COSIT 2005*, Springer, 2005, pp. 1-13.

[15] Google, "Google Earth," URL: http://earth.google.com [accessed May 21, 2008].

[16] Hart, G. and Dolbear, C., "So What's So Special About Spatial?," *Proceedings of the 2006 International Semantic Web Conference*, Springer, 2006.

[17] Hill, L. L., "Core Elements of Digital Gazetteers: Placenames, Categories, and Footprints," *Proceedings of the 4th European Conference on Research and Advanced Technology for Digital Libraries*, Lecture Notes in Computer Science, Vol. 1923, Springer-Verlag, London, 2000, pp. 280-290.

[18] J. Paul Getty Trust, "Getty Thesaurus of Geographic Names (Research at the Getty)," URL: http://www.getty.edu/research/conducting_research/vocabularies/tgn/index.html [accessed December 4, 2007].

[19] Jackendoff, R., "Semantics of Spatial Expressions," *Semantics and Cognition*, MIT Press, Cambridge, MA, 1983, pp. 161-187.

[20] Leidner, J. L., "Toponym Resolution: A First Large-Scale Comparative Evaluation," School of Informatics, University of Edinburgh, Informatics Research Report, July 2006.

[21] Leidner, J. L., Sinclair, G., and Webber, B., "Grounding Spatial Named Entities for Information Extraction and Question Answering," *Proceedings of the HLT-NAACL 2003 Workshop on Analysis of Geographic References*, Vol. 1, Association for Computational Linguistics, Morristown, NJ, 2003, pp. 31-38.

[22] Li, H., Srihari, R. K., Niu, C., and Li, W., "Location Normalization for Information Extraction," *Proceedings of the 19th International Conference on Computational Linguistics*, Vol. 1, Association for Computational Linguistics, Morristown, NJ, August 2002, pp. 1-7.

[23] McGuinness, D. L. and van Harmelen, F. (eds.), "OWL Web Ontology Language Overview," URL: http://www.w3.org/TR/owl-features/ [accessed December 4, 2007].

[24] Merriam-Webster Online, "Dictionary and Thesaurus – Merriam-Webster Online," URL: http://www.merriam-webster.com [accessed May 21, 2008].

[25] Motik, B., Sattler, U., and Studer, R., "Query Answering for OWL-DL with Rules," *The Semantic Web – ISWC 2004: Third International Semantic Web Conference Proceedings*, Springer, 2004.

[26] Open Geospatial Consortium, Inc., "Geography Markup Language | OGC®," URL: http://www.opengeospatial.org/standards/gml [accessed May 21, 2008].

[27] Open Geospatial Consortium, Inc., "Glossary of Terms – W | OGC®," URL: http://www.opengeospatial.org/ogc/glossary/w [accessed May 21, 2008].

[28] Penn Treebank Project, "The Penn Treebank Project," URL: http://www.cis.upenn.edu/~treebank/ [accessed May 21, 2008].

[29] PostgreSQL Global Development Group, "PostgreSQL: About," URL: http://www.postgresql.org/about [accessed May 21, 2008].

[30] Princeton University Cognitive Science Laboratory, "WordNet – Princeton University Cognitive Science Laboratory," URL: http://wordnet.princeton.edu [accessed May 21, 2008].

[31] Randell, D. A., Cohn, A. G., and Cui, Z., "Computing Transitivity Tables: A Challenge for Automated Theorem Provers," *Proceedings of the 11th International Conference on Automated Deduction*, Springer-Verlag, 1992.

[32] Randell, D. A., Cui, Z., and Cohn, A. G., "A Spatial Logic Based on Regions and Connections," *Principles of Knowledge Representation and Reasoning: Proceedings of the 3rd International Conference on Knowledge Representation and Reasoning*, Morgan Kaufman, San Mateo, 1992, pp. 165-176.

[33] Rauch, E., Bukakin, M., and Baker, K., "A Confidence-Based Framework for Disambiguating Geographic Terms," *Proceedings of the HLT-NAACL 2003 Workshop on Analysis of Geographic References*, Vol. 1, Association for Computational Linguistics, Morristown, NJ, 2003, pp. 50-54.

[34] Roth, D. and Yih, W., "Probabilistic Reasoning for Entity & Relation Recognition," *Proceedings of the 19th International Conference on Computational Linguistics*, Vol. 1, Association for Computational Linguistics, Morristown, NJ, August 2002, pp. 1-7.

[35] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y., "Pellet: A Practical OWL-DL Reasoner," *Journal of Web Semantics*, Vol. 5, June 2007, pp. 51-53.

[36] Stanford Natural Language Processing Group, "Named Entity Recognition (NER) and Information Extraction (IE)," URL: http://nlp.stanford.edu/ner/index.html [accessed May 21, 2008].

[37] Stanford Natural Language Processing Group, "The Stanford Parser: A statistical parser," URL: http://nlp.stanford.edu/software/lex-parser.html [accessed May 21, 2008].

[38] U. S. Census Bureau Geography Division, "U. S. Census Bureau – TIGER/Line®," URL: http://www.census.gov/geo/www/tiger/ [accessed November 30, 2007].

[39] Vivid Solutions, Inc., "GMLWriter," URL: http://tsusiatsoftware.net/jts/javadoc/com/vividsolutions/jts/io/gml2/GMLWriter.html [accessed May 21, 2008].

[40] Vivid Solutions, Inc., "JTS Topology Suite," URL: http://www.vividsolutions.com/jts/jtshome.htm [accessed May 21, 2008].

[41] Volz, R., Kleb, J., and Mueller, W., "Towards Ontology-Based Disambiguation of Geographical Identifiers," *WWW 2007: Proceedings of the 16th International Conference on World Wide Web*, Association for Computing Machinery, Banff, Canada, May 2007.

[42] Waldinger, R., Jarvis, P., and Dungan, J., "Using Deduction to Choreograph Multiple Data Sources," *Semantic Web Technologies for Searching and Retrieving Scientific Data*, Sanibel Island, FL, October 2003.

[43] Wessel, M., "Some Practical Issues in Building a Hybrid Deductive Geographic Information System with a DL-component," *Proceedings of the 10th International Workshop on Knowledge Representation meets Databases*, CEUR Workshop Proceedings Vol. 79, September 2003.

[44] Wick, M., Culotta, A., and McCallum, A., "Learning Field Compatibilities to Extract Database Records from Unstructured Text," *Proceedings of the 2006 Conference on Empirical Methods in Language Processing*, Association for Computational Linguistics, Morristown, NJ, July 2006, pp. 603-611.

[45] Zadeh, L. A., "Fuzzy Sets," *Information and Control*, Vol. 8, 1965, pp. 338-353.

[46] Zhou, G. and Su, J., "Named Entity Recognition Using an HMM-Based Chunk Tagger," *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Association for Computational Linguistics, Morristown, NJ, July 2002, pp. 473-480.

[47] Zolin, E., "Description Logic Complexity Navigator," URL: http://www.cs.man.ac.uk/~ezolin/dl/ [accessed May 21, 2008].